J. Parallel Distrib. Comput. 75 (2015) 1–19

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

IMSuite: A benchmark suite for simulating distributed algorithms

Suyash Gupta, V. Krishna Nandivada *

PACE Lab, Department of Computer Science and Engineering, IIT Madras, 600036, India

HIGHLIGHTS

- Genesis and characterization of a new kernel benchmark suite named IMSuite.
- A methodical approach to implement distributed algorithms in task parallel languages.
- Multiple variations of our kernels in two languages X10 and HJ.
- An involved set of input generators and output validators.
- A detailed evaluation of IMSuite.

ARTICLE INFO

Article history:
Received 9 October 2013
Received in revised form
12 May 2014
Accepted 22 October 2014
Available online 28 October 2014

Keywords:
Benchmarks
Distributed algorithms
Performance evaluation
Task parallelism
Data parallelism
Recursive task parallelism

ABSTRACT

Considering the diverse nature of real-world distributed applications that makes it hard to identify a representative subset of distributed benchmarks, we focus on their underlying distributed algorithms. We present and characterize a new kernel benchmark suite (named `IMSuite`) that simulates some of the classical distributed algorithms in task parallel languages. We present multiple variations of our kernels, broadly categorized under two heads: (a) varying synchronization primitives (with and without fine grain synchronization primitives); and (b) varying forms of parallelization (data parallel and recursive task parallel). Our characterization covers interesting aspects of distributed applications such as distribution of remote communication requests, number of synchronization, task creation, task termination and atomic operations. We study the behavior (execution time) of our kernels by varying the problem size, the number of compute threads, and the input configurations. We also present an involved set of input generators and output validators.

1. Introduction

Large distributed applications find their use in a variety of diverse domains: banking, telecommunication, scientific computing, network on chips, and so on. The diverse and complex nature of these distributed applications makes it hard to identify a representative subset of distributed benchmarks. The absence of such a benchmark set hinders the design of new optimizations and program analysis techniques that can be applied uniformly across many distributed applications.

The common denominators of most of the distributed applications are the underlying distributed algorithms. Both the distributed applications and the underlying distributed algorithms display common traits such as communication, timing and failure. We argue that compared to the complex distributed applications, reasoning about these underlying algorithms can be easier and can also help in analyzing the diverse applications that use them. Thus, we believe that a kernel benchmark suite implementing popular distributed algorithms is in order.

We now lay down a set of seven *key* requirements necessary for such a kernel benchmark suite. These requirements are categorized under the following three heads.

(A) *Requirements based on characteristics of kernel benchmarks implementing distributed algorithms*: Our study of popular text books [27,33] and lecture notes [42] on distributed algorithms helped us derive the important characteristics of typical distributed algorithms; these characteristics form the basis of our first three key requirements.

1. The algorithms implemented by the kernel benchmarks must solve common challenges in distributed systems.
2. The kernels should cover important characteristics of distributed systems such as communication (broadcast, unicast, or multicast), timing (synchronous, asynchronous or partially synchronous) and failure.

* Corresponding author.
E-mail addresses: suyash@cse.iitm.ac.in (S. Gupta), nvk@cse.iitm.ac.in (V. Krishna Nandivada).

http://dx.doi.org/10.1016/j.jpdc.2014.10.010
0743-7315/

Problem	Applications and/or domains of interest.
Breadth first search (BFS)	• community analysis: community detection algorithm repetitively calls BFS to discover all the community structures. • used in Routing Information Protocol (RIP) to calculate shortest path between routers and nodes. • Velvet [44] (manipulates de Bruijn graphs for genomic sequence assembly) uses BFS to remove redundant paths created by the biological variants.
Consensus	• checking reliability of a distributed system with byzantine nodes. • distributed detection of the byzantine sensors and minimum number of byzantine sensors to prevent detection. • fault tolerant MapReduce for critical data processing to detect and handle byzantine faults [9].
Routing table	• OSPF protocol to compute shortest path for each node. • Ecological route search for reducing fuel consumption [23].
Dominating set (DS)	• efficient router placement in mobile adhoc networks using node clustering. • protein-protein interaction (PPI) networks can compute DS to determine biologically central genes that play a topologically central role [29].
Maximal independent set (MIS)	• clustering in ad-hoc radio and sensor networks for initializing the network. • identifying the components in the context of symmetry breaking. • For analysis of protein-ligand binding – MIS can be used to separate the redundant protein structures from the similarity graph [41].
Committee creation	• token (message, gossip) dissemination in edge-dynamic networks (wireless ad hoc, sensor) and resource constrained and unreliable networks. • to compute the upper bound on the message details to be sent, in adversarial dynamic graphs [24].
Leader election	• selecting a coordinating node for a network. • replacing an erroneous central lock-coordinator in distributed databases. • large-scale climate simulation analysis using virtual machines (VM)–leader election helps in coordination of the VM [19].
Spanning tree	• present in IEEE 1394.1 draft standard for interconnecting LANs, to maintain the spanning tree during dynamic addition and deletion of bridges. • detection of spread of toxins in population. • to measure the mass segregation in star clusters by calculating length of minimum spanning tree for various star clusters [1].
Graph coloring	• coloring growth bounded graphs such as unit disk graphs. • frequency or time slots assignment in wireless network. • approximating the Schur complement using graph coloring matrices [37].

Fig. 1. Core distributed computing problems and their applications (see [1,9,19,23,24,29,37,41,44]).

3. The benchmark kernels should simulate the behavior of distributed systems consisting of (partially) independent nodes and the interconnect thereof.

(B) *Requirements based on the target hardware*:

4. The execution of the kernels implementing distributed algorithms should not necessarily require a complex hardware setup; these should be usable in the presence of a shared memory multicore/distributed memory multicore or even a sequential system.

(C) *Requirements based on best practices in existing benchmark suites*: The final two requirements are derived from the best practices followed in well known benchmark suites, such as PBBS [36], NPB [4], BOTS [14], PARSEC [5].

5. The kernels should be small in size and easy to debug.
6. The benchmark suite should provide a variety of inputs (with varying configurations and sizes) and convenient means to verify the generated output.
7. The benchmark suite should provide means to analyze static and dynamic characteristics specific to the domain under consideration (distributed systems in our case).

Our study of existing benchmark suites [36,4,14,5,6,2,3,7,10,13,20,26,31,38,43] has found that none of them meet majority of the aforementioned key requirements. Our goal is to design a benchmark suite that meets all our stated requirements. As a first step, we shortlist a set of important problems in the context of distributed systems.

Fig. 1 shows some of the *core* problems in the area of distributed computing and lists a few of their extremely diverse applications. The centrality of these problems can also be seen from the importance given to them in popular textbooks and lecture notes on distributed algorithms [27,33,42]. In this paper, we present and characterize a new kernel benchmark suite named `IMSuite`: *IIT Madras benchmark suite for simulating distributed algorithms* that implements some of the classical algorithms to solve these core problems; we refer to these algorithms as the *core* algorithms.

`IMSuite` implements the core algorithms in two task parallel languages X10 [34] and HJ [8]. X10 and HJ languages with their APGAS-model to easily simulate the distributed systems, light weight tasks to represent the computation in the distributed nodes, and clocks/phasers to model lock step synchrony in irregular and recursive applications, give a convenient way to program distributed kernels. One of the main advantages of using these languages is that they can easily simulate a large set of distributed nodes even in the absence of complex distributed hardware. Even though the initial uses of X10 and HJ languages were mainly in the scientific domain, there is increasing interest to use these languages in other domains. For example, ScaleGraph [11] is an X10 library which can be used for analyzing very large (billion scale) graphs. Similarly, XGDBench [12] provides a platform for storing large graph databases in exascale cloud. This trend conforms to the general trend in the HPC domain where the community has started

taking increasing interest in irregular computations involving distributed applications. Some popular examples include Ligra [35], Pregel [28], Pegasus [22], PowerGraph [16] and Knowledge Discovery Toolkit [25].

Our contributions

- We present a study of a large set of existing benchmark suites and discuss their limitations with respect to our stated key requirements (Section 2).
- We give a methodical approach to implement distributed algorithms in task parallel languages to run on hybrid systems[1] (Section 4).
- Considering the different popular parallel programming styles, we present multiple variations of our kernels in both X10 and HJ. These variations (totaling 31 per language) can be broadly categorized under two heads: (a) *Varying synchronization primitives*: Our benchmark kernels can use fine grain synchronization primitives (such as phasers in HJ and clocks in X10), or can realize synchronization by joining/terminating each task and recreating them later. (b) *Varying forms of parallelization*: `IMSuite` contains a data parallel implementation for each core algorithm. Further, `IMSuite` also includes recursive task parallel versions for some of the core algorithms. Besides these parallel versions `IMSuite` also includes the corresponding serial implementations (Section 5).
- We provide an algorithm specific input generator that can generate a variety of inputs with varying configurations. Each benchmark also includes an output validator.
- We characterize `IMSuite` on a hybrid system. Our characterization covers interesting aspects of distributed applications such as distribution of remote communication requests, number of synchronization, task creation, task termination and atomic operations. We study the behavior (execution time) of our kernels by varying the problem size, the number of compute threads, and the input configurations. (Section 6).

2. Related work

In this section we categorize some of the popular benchmark suites catering to parallel and distributed systems and discuss their limitations with respect to our stated key requirements.

Applications vs. kernels vs. micro-kernels—Many of the well-known benchmarks consist of a set of representative applications. Examples include NPB [4], BOTS [14], PARSEC [5] (including its two prior *avatars* SPLASH [38] and SPLASH-2 [43]), BenchERL [2], SPEC-OMP [3], JGF [10], NGB [13], SPEC-MPI [31] and HPCC [26]. While PARSEC, JGF, NPB and BenchERL also contain a few kernels, benchmark suites like PBBS [36] focus only on kernel benchmarks. Similarly, JGF contains a few micro-kernels as well, while EPCC [7,6] and IntelMPI [20] contain only micro kernels. Compared to the application-oriented benchmarks, the kernel benchmarks are small in size, simpler to understand, easier to debug and provide insights on how a certain algorithm behaves. Micro-kernels on the other hand are helpful to study a specific feature of a language, runtime or architecture.

Scientific vs. non-scientific—Most of the parallel benchmark suites target scientific or mathematical computations. Examples include BOTS, JGF, HPCC, NPB, SPEC-OMP, SPEC-MPI and PARSEC. The PBBS benchmark suite consists of a mixed bag of scientific and graph computations. Benchmark suites like BenchERL, EPCC, and IntelMPI consist of mainly synthetic benchmarks.

Task parallel vs. loop parallel vs. recursive—BOTS and PBBS admit both task parallel and recursive task parallel computations. In contrast, JGF, HPCC, SPEC-OMP, SPEC-MPI, IntelMPI, PARSEC, NPB, NGB, and EPCC suites include computations chiefly depicting loop level parallelism.

Parallel vs. hybrid systems—Benchmark suites like NPB, JGF, NGB, BenchERL, SPEC-MPI, EPCC, and IntelMPI contain benchmarks that can run over hybrid systems, while rest of the benchmark suites can run only on a parallel system.

Of the seven key requirements discussed earlier in the section, the first four are specific to the distributed applications and hence are not satisfied by any of the discussed benchmark suites. The PBBS benchmark suite satisfies Req#5 and Req#6. On the other hand, PARSEC, JGF, NPB and BenchERL have a mix of small kernels and large applications, and satisfy Req#5 partially. Similarly, benchmark suites such as BOTS, BenchERL, HPCC and SPEC include an output verifier for a pre-defined input, and satisfy Req#6 partially. Req#7 is partially satisfied by PBBS, PARSEC, SPLASH and BOTS—they allow the user to measure some dynamic characteristics that are pertinent to parallel programs (such as, number of tasks, barriers, joins, and so on).

Compared to these benchmark suites, `IMSuite` satisfies all the key requirements. It consists of kernels that implement popular distributed algorithms (mostly graph based) that are mainly irregular in nature and can be used in both scientific and non-scientific computations. The kernels in `IMSuite` exhibit both loop and recursive task parallelism. While the current implementation of `IMSuite` is in X10 and HJ, these kernels can easily be ported to other languages that support appropriate runtime models (such as APGAS or global address space).

3. Background

3.1. Core algorithms

Considering the importance and popularity of the problems discussed in Fig. 1, it is not surprising to find a plethora of algorithms in the literature that solve these problems. However, a benchmark suite can accommodate only a small number of algorithms for each of these problems. Many different schemes can be used to select such a subset, each with its own set of challenges: most recent (challenge: this set will keep changing all the time); most popular/efficient (challenge: hard to find an agreement in the community and keeps changing with time); classical algorithms (challenge: may not be recent). Considering pedagogy and popularity we choose the last scheme.

We now briefly describe some of the classical algorithms that solve the problems discussed in Fig. 1. Fig. 2 presents some characteristics of these algorithms.

Breadth First Search (BFS): We use two different BFS algorithms *BF* [42] and *DST* [42]. While *BF* outputs the distance of every node from the root, *DST* outputs the BFS tree.

Byzantine agreement: The byzantine agreement (*BY*) algorithm [30] builds a consensus among the "good" nodes of a network that may also contain "faulty" nodes.

Routing: In the Dijkstra routing (*DR*) algorithm [40] each node in the network works independently and computes a routing table in parallel.

Dominating set: The dominating set (*DS*) algorithm [42] creates a dominating set using a probabilistic method that depends on the first and second level neighbors of a node.

***k*-Committee**: For a given integer value of *k*, the *k*-committee (*KC*) algorithm [42] partitions the input nodes into committees of size at most *k*.

[1] A hybrid system may consist of one or more distributed nodes (with a capability to communicate with each other), each node may consist of one or more cores, and each core in turn may have one or more hardware threads.

Problem Category (Abbr)	NW type	Time Complexity	Message Complexity
Breadth First Search (*BF*)	General	$O(D)$	$O(nm)$
Breadth First Search (*DST*)	General	$O(D^2)$	$O(m + nD)$
Consensus(Byzantine) (*BY*)	General	$O(D)$	$O(n^2)$
Routing Table Creation (*DR*)	General	$O(n^2)$	$O(nm)$
Dominating Set (*DS*)	General	$O(\log^2 \Delta \times \log n)$	$O(n \times \Delta^2 \times \log^2 \Delta \times \log n)$
Maximal Independent Set (*MIS*)	General	$O(\log n)$	$O(m \log n)$
Committee Creation (*KC*)	General	$O(K^2)$	$O(K^2 m)$
Leader election (*DP*)	General	$O(D)$	$O(Dm)$
Leader election (*HS*)	Ring (bi)	$O(n)$	$O(n \log n)$
Leader election (*LCR*)	Ring (uni)	$O(n)$	$O(n^2)$
Spanning Tree (*MST*)	General	$O(n \log n)$	$O(m \log n)$
Vertex Coloring (*VC*)	Tree	$O(\log^* n)$	$O(n \log^* n)$

Fig. 2. Core algorithms and their characteristics. Notation: n denotes the number nodes, m denotes the number of edges, and D denotes the diameter, K denotes the maximum committee size and Δ denotes the maximum degree of the graph.

Syntax	Explanation
`[clocked] async {S1}`	spawns a new asynchronous task to execute the statement `S1`. The `clocked` option registers the task on the set of clocks held by the current task.
`[clocked] finish {S1}`	waits for all the tasks created within `S1` to terminate. The `clocked` option introduces a new clock that the task executing `S1` gets registered to.
`atomic {S1}`	updates the shared data in `S1` in an atomic fashion, provided other possible accesses to that shared data also happens inside an `atomic`.
`Clock.advanceAll`	a blocking call that advances all the clocks the current task is registered with. It acts as a barrier.
`x = at(p) {S1}`	executes the expression `S1` at place `p` and `x` stores the return value.
`var R: Region; R = 0..(n-1)`	creates a region R containing n elements: `0...n-1`.

Fig. 3. X10 command cheat sheet.

Maximal independent set: *MIS* [42] uses a randomized algorithm to compute the maximal independent set for a given input graph.

Leader election: We consider three different leader election algorithms. The *LCR* and *HS* algorithms [27] work on a set of nodes organized in a ring network, where the data flow is unidirectional and bidirectional, respectively. Compared to that, the *DP* algorithm [32] works on a set of nodes organized in any general network.

Minimum spanning tree: The *MST* algorithm [42] works on a weighted graph. It starts by marking every node as an independent fragment, and proceeds by joining fragments along the *minimum weighted edge*, till a lone fragment is left.

Vertex coloring: The vertex coloring (*VC*) algorithm [42] colors the nodes of a tree with three colors. It first colors the tree using six colors using a fast algorithm $O(\log^* n)$ and then uses a *shift down* operation (constant time) to color the tree using three colors.

The popularity of the chosen algorithms can be seen from the large number of works (books and papers) that cite (use and/or extend) them.

3.2. X10 and HJ background

Fig. 3 presents some constructs of X10 relevant to this manuscript (see the language manual [34] for details).

We use `async` to spawn a new task, `finish` to join tasks, and `atomic` to provide mutual exclusion. X10 provides an abstraction of a `Clock` that helps tasks make progress in lock step synchrony. A task may be registered on one or more clocks and all the tasks registered on a clock make progress in lock step by *advancing* the clocks. A clock is considered to have advanced to the next "clock tick" if all the tasks registered on that clock have requested the advancement of the clock.

In X10, a `place` abstracts the notion of computation (multiple tasks) and data (local to the place). The set of places available to a program are fixed at the time the program is launched. The `at` construct can be used to access remote data.

A *region* is used to represent the iteration space of loops and the domain of arrays. A *distribution* maps the elements of a region to the set of runtime places.

We define two subsets of X10: (a) `X10-FA`—uses the `finish`, `async` and `atomic` constructs for task creation, join and mutual exclusion, respectively. Synchronization is achieved by joining/terminating all the tasks and recreating them later. (b) `X10-FAC`—uses the abstraction of *clocks* in addition to the constructs of `X10-FA`. Clocks provide efficient synchronization primitives that can be used to yield arguably more compact and efficient programs.

Comparison with HJ: The parallelism related constructs of Habanero Java [8] are similar to that of X10 with minor differences in syntax and semantics. For example, HJ constructs `isolated`, `next`, and `phaser` map to corresponding X10 constructs `atomic`, `Clock.advanceAll` and `clock` (refer to the HJ manual [18] for details). Similar to the two subsets of X10, we define two corresponding subsets for HJ: `HJ-FA` and `HJ-FAP`.

4. Transformation scheme

We now present an overview of our scheme for implementing distributed algorithms in a task parallel language, to be executed on a hybrid system. We list some of the main abstractions pertinent

to distributed algorithms and lay down a procedure for their implementation.

Node: A *node* in a distributed algorithm requires some data (such as a unique identifier, a mailbox, information about neighbors and so on) and performs some computations.

The computation of a node can be abstracted by one or more parallel tasks, in task parallel languages. A distributed node (including both computation and data) can be abstracted by an X10 *place* running only the task(s) corresponding to that node; we refer to it as the *Unique-Place (UP) model*. Another model in which a distributed application can be simulated is one where all the nodes of the system are simulated at a single place; we term it as the *Single-Place (SP) model*. This simulates a particular type of distributed system where the inter-node communication cost is minimal. We can also consider a more general scenario, where the runtime consists of multiple places and each place may simulate the tasks corresponding to more than one node; we term it as the *Multi-Place* (MP) *model*.

Communication: A set of distributed nodes communicate with each other through message transfer. These messages are transferred, from one node to another, along the links of the underlying network.

Our simulation of the transfer of data between two connected nodes of a network depends on the runtime model (MP, UP or SP model). In the context of an SP model, the data transfer is done using the shared memory. On the other hand, in the context of MP and UP models, the data transfer may involve message passing.

Timing: In a distributed system, nodes can work asynchronously or synchronously. In a synchronous setting there is an assumption of existence of a global clock and the nodes proceed in a lock-step fashion, synchronized over the global clock. Contrast to that, in an asynchronous setting there is no concept of a global clock and each node works independently.

We achieve lock step synchrony by using fine grain synchronization primitives (such as phasers in HJ and clocks in X10) or by repeated task-join and task-creation operations (see Section 4.1 for an example).

Phases and rounds: Distributed algorithms are organized around the notion of phases and rounds; each phase consists of one or more rounds. Phases and rounds can be implemented using serial loops.

Messages and mailbox: Nodes in a distributed system communicate by exchanging messages. The size and structure of a message depend on the underlying algorithm. Each message is delivered to the receiver's mailbox (a FIFO queue). The design of the mailbox must ensure that it can hold all the messages required at any point of time.

4.1. Sample transformation

In this section we illustrate our transformation scheme using an example. Fig. 4 presents the core of the *LCR* algorithm (see Section 3.1). Each node j contains three fields: uid_j (the unique identifier of the node), $send_j$ (identifier of the leader as per node j—initialized to uid_j), and $status_j$ (if it is a leader or a common node). The *LCR* algorithm runs for n rounds. In each round every node j sends $send_j$ to its neighbor (successor) and receives the incoming message from its predecessor. Since *LCR* algorithm works on a unidirectional ring network, a node can at most receive one message per round. This allows us to set the size of the mailbox of each node to one.

We now briefly explain how we derive a benchmark kernel for the *LCR* algorithm. For illustration purposes we use X10-FAC as the target language and later state the differences between a code written in X10-FAC and in X10-FA. The structure of the *Node* class for our implementation of the *LCR* algorithm is shown in Fig. 5. We note two interesting points: (a) the mbox field can hold at most one message, (b) the nextIndex field can be eliminated by some smart design decisions (e.g. in a unidirectional ring network the nextIndex of the jth node can be set to $j + 1$).

Input n nodes each having a unique uid; **Output** Leader
for $round \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n *nodes* in parallel **do**
    transmit $send_j$ to its clockwise neighbor ;
    $x \leftarrow$ incoming message;
    if $x > uid_j$ **then** $send_j \leftarrow x$;
    else if $x = uid_j$ **then** $status_j \leftarrow$ leader;
    else do nothing ;

Fig. 4. Distributed leader election *LCR* algorithm.

```
class Node {
      var uid:Int;
      var mbox:Int;          // mail box
      var nextIndex:int;     // neighbor index
      var status:boolean;    // true => leader
      var send:Int;          // outgoing message
      var leader:Int; }
```

Fig. 5. Structure of the abstract node for LCR.

```
leader_elect_lcr(val n:Int){
   var R: Region = 0..(n-1) ;
   var D: Dist = Dist.makeBlock(R);
   var ndSet: DistArray = DistArray.make[Node](D);
   for(var round:Int=0; round<n; round++) {
      clocked finish {
         for(j in D){ // run each node
            clocked async at(D(j)){ // in parallel
               Transmit(ndSet(j).send, nextIndex);
               Clock.advanceAll() ;
               if(ndSet(j).mbox > ndSet(j).leader) {
                  ndSet(j).send = ndSet(j).mbox;
                  ndSet(j).leader = ndSet(j).mbox; }
               elseif(ndSet(j).mbox==ndSet(j).uid) {
                  ndSet(j).status = true;
                  ndSet(j).leader = ndSet(j).uid;
               } } } } } }
```

(a)

```
Trasmit(val receiver:Point, val msg:Int){
   at(D(receiver)) ndSet(receiver).mbox=msg;}


leader_elect_lcr(val n:Int){
   ...
   for(var round:Int=0; round<n; round++){
      finish{
         for(j in D){ // run each node
            async at(D(j)){ // in parallel
               Transmit(ndSet(j).send, nextIndex);
            } } }
      finish{
         for(j in D){ // run each node
            async at(D(j)){ // in parallel
               if(ndSet(j).mbox > ndSet(j).leader){
                  ... }
               elseif(ndSet(j).mbox==ndSet(j).uid){
                  ...
            } } } } }
```

(b)

Fig. 6. (a) Core of the LCR algorithm in X10-FAC; (b) core of the LCR algorithm in X10-FA; only differences with respect to the X10-FAC version are shown. The X10 specific constructs are shown in **bold**. See Section 3 for X10 syntax.

Fig. 6(a) shows the core of the *LCR* algorithm in X10-FAC. It creates a blocking distribution D over a region R (of n points, where n = number of nodes) and allocates the array ndSet (of n elements), distributed over D. The number of blocks in the distribu-

tion D is set to the number of *places* at runtime. In each round, the parallel task corresponding to each node transmits its message and waits (by using `Clock.advanceAll`) for the message from its neighbor. After that, each task recomputes the leader related information based on the received message and proceeds to the next round.

Fig. 6(b) sketches the `X10-FA` kernel for *LCR*, showing only the differences with respect to the `X10-FAC` kernel shown in Fig. 6(a). This `X10-FA` version uses repeated task-join and task-creation operations to synchronize the tasks corresponding to the nodes. `X10-FAC` implementation can be considered lightweight as it uses fewer number of task creation (*async*) and join (*finish*) operations and utilizes the lightweight synchronization operations offered by *Clocks*.

Compared to *LCR*, where the `X10-FA` and `X10-FAC` implementations are not much different, there are other kernels (such as *DS*) where the differences are significant. This is especially true when the clock based synchronization operations are nested deep inside conditional or looping constructs.

5. Internals of IMSuite

In this section, we briefly explain the internal details of `IMSuite`. This benchmark suite implements the twelve core algorithms described in Section 3.1. Considering the different popular parallel programming styles, we have implemented multiple variations of these algorithms:

Varying the synchronization primitives: We implement all our variations in two subsets of X10: `X10-FA` and `X10-FAC`. All the core algorithms (except *DR*) have been implemented in both `X10-FA` and `X10-FAC`. In case of *DR*, we found no scope of using low level synchronization primitives like clocks. And hence we have this algorithm implemented only in `X10-FA`.

Varying forms of parallelization: For each of the core algorithms implemented in `X10-FA` and `X10-FAC` we present a data parallel implementation. For five of the core algorithms (*BF, DST, BY, DR* and *MST*) implemented in `X10-FA`, we also provide variations that exploit recursive task parallelism. Further, for three of these algorithms (*BF, DST* and *MST*) we have efficient implementations that use clocks (implemented in `X10-FAC`).

Along with the above discussed variations, we can also vary the runtime model (`MP`, `UP`, or `SP` model) by setting the number of places to be a divisor of the input (for `MP` model), or to the input size (for `UP` model), or to one (for `SP` model).

For each of the core distributed algorithms we also present a serial implementation in X10. The serial implementations do not create any parallel tasks—they simulate the behavior of the parallel nodes by serializing their execution in a predefined order. Similar to their parallel counterparts, the runtime behavior of the serial programs can be controlled by varying the underlying runtime model (`MP`, `UP` or `SP`). An interesting point to note is that a serial program whose data is distributed over partitioned global address space mimics a distributed system partly—where accessing remote data is more expensive than accessing the local data.

In summary, we provide a set of 35 (12 in `X10-FA`, 11 in `X10-FAC`, and 12 serial) iterative kernels and 13 (5 in `X10-FA`, 3 in `X10-FAC`, and 5 serial) recursive kernels; 48 kernels in total.

Considering the growing popularity of HJ, we have also implemented these 48 kernels in `HJ-FA` and `HJ-FAP` subsets of HJ (similar to the variations provided by `X10-FA` and `X10-FAC`). Owing to the current limitations of the HJ runtime, these kernels can only be simulated at a single place. Thus, we can only realize the `SP` runtime model here.

Considering the possibility that in practice these core algorithms may do some more computation in addition to that specified by the algorithm, all our kernel benchmarks take an additional option to introduce a user specified workload in each asynchronous task. Currently, we present a naive workload function that injects a series of arithmetic computations (quantity specified by the user at runtime) to each asynchronous task in the kernel. We can foresee a workload function with additional characteristics, such as one that pollutes the L1/L2 cache, or one that introduces additional packets in the network and so on. The design of such sophisticated workload functions is left as future work.

5.1. Input generator

The input to all the `IMSuite` kernels is an abstraction of a distributed system consisting of the details about its configuration (for example, nodes, edges, weights and so on). `IMSuite` comes with a set of input generators that generate inputs specific to each kernel benchmark. Depending on the core algorithm under consideration each input generator admits a set of options that can be used to tune the generated input. Some of the common options are the number of nodes in the distributed graph (referred as the *size* of the input), the type of the graph (complete, sparse and so on), weights of the edges and so on. Our input generators use a random number generator to generate the details (such as weights, adjacency information, unique identifiers of the nodes, and so on) of the distributed graph. To make the input generation process deterministic, our input generators optionally take a seed (default value set to the prime number[2] 101). The users of `IMSuite` are required to specify the seed used to generate their input; this can help users to communicate their findings in a more meaningful manner. Each input generator is serial in nature and is written in Java.

The different types of graphs generated by our input generators depend on the target algorithm: ring for *LCR* and *HS*, tree for *VC*, and any arbitrary graphs for others. Considering the typical configurations of trees and arbitrary graphs, our input generators admit additional options (described below).

Trees: We allow three topologies for trees based on the typical usages: *Star*, *Chain* and *Random*. The last one takes an additional input that specifies the maximum degree for any node. The choice of Star and Chain as two predefined topologies stems from the behavior of the *VC* algorithm. For a fixed input size, *VC* takes the maximum time for a Star topology and minimum for a Chain.

Arbitrary graphs: Our input generators can generate three types of arbitrary graphs: (i) complete graphs (to help realize fully connected networks), (ii) sparse graphs (most prevalent form of graphs in practice, for example, web graph, internet topology graph, social network, cloud network and so on) and (iii) random graphs, where the edges are chosen at random.[3] The limiting cases of the sparse graphs (with edges $n - 1$ and $n \log n$, where n is the number of nodes) are present as two special options named *SP-Min* and *SP-Max*. To enable the comparison between these two limiting cases, our input generator ensures that the edge set of *SP-Max* variation is a superset of the edge set of *SP-Min*.

5.2. Output validators

Each kernel also consists of an output validator to validate its output. The output validator assumes that it has access to the complete input and output and may reuse some internal data structures of the main program, for efficiency reasons. The output validators are serial in nature and are not timed.

5.3. Conformance to the key requirements

We now discuss how the `IMSuite` kernels conform to the key requirements specific to distributed systems (Req#1–#3). These

[2] A set of interesting anecdotes about the number 101 can be found here: http://primes.utm.edu/curios/page.php?short=101.

[3] Szemerédi Regularity Lemma [39] indicates that properties of dense networks can be studied using random graphs.

Name	#Lines of Code		#Static		#Lines of Code		#Static		
	X10-FA	HJ-FA	Fin	Mut	X10-FAC	HJ-FAP	Fin	Bar	Mut
BF	330	320	2	1	330	310	1	1	1
DST	510	500	7	3	500	490	5	2	3
BY	510	460	3	1	505	455	2	1	1
DR	370	355	1	0	-	-	-	-	-
DS	650	600	9	2	580	510	1	8	2
KC	490	490	10	2	480	470	7	3	2
DP	440	395	4	1	430	375	1	3	1
HS	440	435	5	0	430	420	3	2	0
LCR	260	245	3	0	250	240	2	1	0
MIS	380	330	5	3	365	310	2	3	3
MST	880	790	15	4	850	760	8	9	4
VC	420	395	5	0	410	390	3	2	0

(a) Iterative kernels.

Name	#Lines of Code		#Static		#Lines of Code		#Static		
	X10-FA	HJ-FA	Fin	Mut	X10-FAC	HJ-FAP	Fin	Bar	Mut
BF	275	275	2	1	270	270	1	1	1
DST	480	475	6	3	475	470	5	1	3
MST	705	630	11	3	675	590	4	7	3
BY	425	440	3	2	–				
DR	375	360	3	0	–				

(b) Recursive kernels.

Fig. 7. Static characteristics of IMSuite kernels.

kernels are derived from the algorithms that solve some of the core problems discussed in Fig. 1—Req#1. The IMSuite kernels cover the important aspects of distributed systems, such as communication (unicast: *LCR*, broadcast: *BY* and *DR*, multicast: rest all); timing (synchronous: *DP*, *HS*, *LCR* and *VC*, asynchronous: *DR* and the recursive kernels of *BY*, and partially synchronous: rest all); and failure: the *BY* kernels admit "nodes" that may fail (faulty nodes)—Req#2. Our kernels take as input an abstraction of a set of (partially) independent nodes and their interconnect. By varying the input type we can realize varied interconnects—Req#3.

6. Evaluation

We present the characterization of IMSuite on an IBM cluster consisting of four hardware nodes.[4] Each hardware node of the cluster has two Intel E5-2670 2.6 GHz processors, each processor has eight cores and each core can make use of (up to) two hardware threads. Thus, we can have up to 128 dedicated hardware threads for our simulations. Each core has its own local L1 cache that is shared by the two hardware threads. The two hardware nodes are connected by an FDR10 Infiniband interconnect. Our chosen hardware configuration helps us study the impact of multiple hardware threads, multiple cores, multiple processors, and multiple hardware nodes, while simulating different distributed kernels. For our simulations we use x10-2.3.0-linux x86 version of X10, jdk1.7.0_09 version of Java and hj-1.3.1 version of HJ.

For evaluating the X10-FA and X10-FAC kernels, we use the X10 compiler targeting the Java backend (a.k.a. the Managed X10) and the runtime uses sockets for supporting inter-hardware-node communication. X10 provides multiple implementations of the scheduler for the asynchronous tasks, such as work stealing and fork-join scheduler (*default*). Similarly, HJ also allows the use of multiple scheduling algorithms, such as work-sharing blocking (*default*), work-sharing cooperative, workstealing help-first, work-stealing work-first, and work-stealing adaptive. For the purpose of this evaluation, we limit ourselves to the default options.

Our characterization involves, among other things, analyzing the behavior of the IMSuite kernels with varying number of available hardware threads (HWTs). Our hardware configuration directs the way we increase the HWTs for our experiments: 1/2/4/8 HWTs correspond to one/two/four/eight independent cores on a processor; 16 HWTs correspond to all the cores present in a hardware node; 32 HWTs correspond to all the cores in a hardware node running two hardware threads each; 64 HWTs correspond to 32 HWTs on each of the two hardware nodes; and 128 HWTs correspond to 32 HWTs on each of the four hardware nodes.

We use the results of the insightful paper of George et al. [15] and compute the average running time for our kernels after executing each of them for 30 times. This helps in reducing the noise in the results arising due to many non-deterministic factors common in a Java based runtime (for example, thread scheduling, garbage collection and so on).

Considering the fact that many real life network/distributed systems are sparsely connected, we restrict our evaluation to sparse networks. Specifically, we focus on the limiting cases of sparse inputs: *SP-Max* and *SP-Min*. Similarly for *VC*, we use the two corresponding limiting case inputs (*Star* and *Chain*). We believe these limiting cases will give us a good understanding of how the benchmarks may behave for other intermediate inputs. We refer to these limiting case inputs as Mx-In and Mn-In, respectively. However, *HS* and *LCR* work on ring networks and entertain no such variations in the network configuration (that is, Mx-In = Mn-In).

6.1. Kernel characteristics

In this section, we discuss some of the static characteristics of our kernels. Fig. 7(a) and (b) present these characteristics for the iterative and recursive kernels, respectively. In these tables, Mut is used as a generic name referring to mutex operations—atomic construct in X10 and isolated construct in HJ. Similarly,

[4] To avoid the confusion between the hardware nodes and the abstraction of nodes in the input, we explicitly qualify the nodes in the hardware as "hardware nodes". We use the generic term "nodes" to denote the input nodes.

Bar is used as a generic name referring to barrier operations—Clock.advanceAll() construct in X10 and next in HJ. We use Fin to abbreviate the finish operations.

It can be seen that all the kernels in IMSuite are relatively small in size; their sizes vary approximately between 200 and 900 lines. Further, the number of static finish, mutex and barrier statements are quite small. However, their dynamic counts vary depending on the number of actual input. Interested reader can refer to Appendix for a discussion on the dynamic characteristics of IMSuite kernels.

6.2. Performance analysis

In this section we study the effect of three *key* parameters on the behavior of IMSuite kernels. These key parameters are: (a) number of available hardware threads (HWTs), (b) input size (denoting the number of nodes), and (c) number of node clusters.[5] Variations in the number of node clusters are achieved by varying the number of runtime places in the X10-FA and X10-FAC kernels. We study the effect of these parameters both in isolation (by varying only one parameter and fixing the rest) and in conjunction with each other (by varying two or three parameters at a time and fixing the rest). Varying multiple parameters at the same time may lead to an overly large number of experimental points. We handle this situation by varying these parameters in "sync" (all the varying parameters get the same value, for a given evaluation point). For fixing the first two key parameters we use the following (arbitrary) policy: when the key parameter "input size" is fixed, we set it to 64 nodes; when the key parameter "HWTs" is fixed, we set it to 8. However, for fixing the third key parameter ("number of node clusters"), we either set it to 1 (to evaluate the HJ-FA and HJ-FAP kernels), or fix it in sync with the other key parameters. For the purpose of this study, we set the input type to Mx-In. Later (Section 6.2.8), we also analyze the effect of input type (Mx-In and Mn-In) on the behavior of the IMSuite kernels.

Limits for the key parameters of our study: We vary the input size between 8 and 512 nodes. This range of inputs allows us to model mini (8–32), small (64–128) and medium (256–512) scale clusters. The execution times were too insignificant for inputs smaller than 8 nodes and it took too long (of the order of several days) to execute programs with inputs larger than 512 nodes. Our choice of input sizes conform to sizes of distributed systems used in recent times. For example, Jabeen et al. [21] utilize 25–120 nodes for small setup and 166–400 nodes for a larger setup. Similarly Gui et al. [17] varied the number of nodes in the range of 20–120.

We vary the number of HWTs between 1 and 128 (limited by the experimental system at hand) when using the X10 runtime. The absence of a distributed HJ runtime limits the maximum number of HWTs, for running our HJ based kernels, to 32 on our hardware. We vary the number of clusters between 1 and 128.

6.2.1. Effect of varying the number of HWTs (input size and number of clusters fixed)

Fig. 8(a) and (b) present the execution time (speedup) statistics of the X10-FA and X10-FAC kernels, for varying number of hardware threads in multiples of two. For all these runs we set the input variation to Mx-In and input size to 64 nodes, which in turn fixes the maximum of HWTs to 64. We have also studied the behavior of these kernels for the Mn-In input variation and have found it to be similar. We plot the execution time of the kernels with respect to that of the serial implementation in the UP model. These plots show that the overall performance for all the kernels improves with increase in the number of available HWTs. However, for any specific kernel the quantum of improvement varies with the chosen input and the number of available HWTs and their configuration.

The performance improvement is more or less linear when we increase the number of hardware cores from 1 to 8 (intra-processor communication only)—this improvement is because of the increased sharing of workload among the HWTs. On moving to 16 HWTs there is a slight dip in performance improvement (compared to 8 HWTs), owing to the inter-processor communication that comes into picture. The performance improvement on 32 HWTs, compared to 16 HWTs is much less. In case of 32 HWTs, while it doubles the sharing of workload by HWTs, it does not double other resources (for example, L1 cache). As a result it incurs additional overheads due to increased conflicts in accessing shared resources such as cache, interconnect, and so on. This behavior is quite pronounced in *HS* where it leads to a slight dip in performance (compared to 16 HWTs). On going from 32 to 64 HWTs, the performance improvement depends on a host of factors—increased communication cost (inter-hardware-node communication is more expensive than intra-node), decreased scheduling overheads (each place runs on a unique HWT), decreased resource conflicts. Depending on the specific kernel the effect varies. For example, in Fig. 8(b), *HS* shows 13% improvement and *BF* shows 96% improvement.

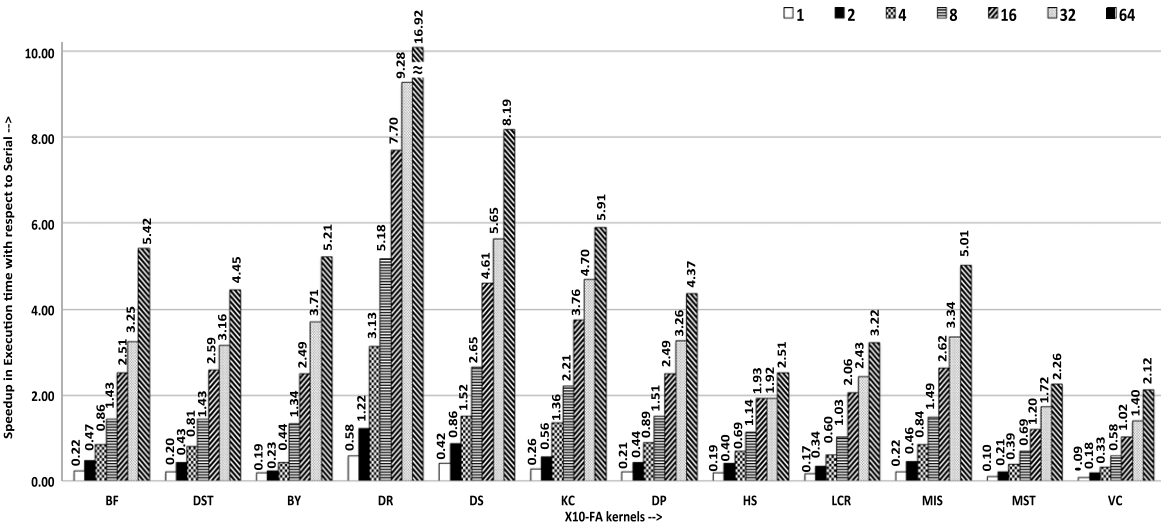
An interesting point to note is that in general for fewer hardware threads (1, 2, 4), the serial versions in the UP model run faster than the X10-FA and X10-FAC versions. This is due to the additional task creation, scheduling and termination overheads present in these kernels. As we increase the number of hardware threads (8, 16, 32, 64), the task scheduling overheads decrease, and the effect of increased workload sharing starts dominating the above mentioned overheads.

As discussed in Section 5, our kernels admit an additional option to introduce a user specified workload in each asynchronous task. We found that such an option is especially useful when our HJ based kernels are simulated (on SP model), where the time taken to execute these kernels is too small (of the order few tens of milliseconds) to reason about the behavior of these benchmarks; we tested the benchmarks for input size of 64 nodes. To overcome this issue, we set a moderate workload of 10 million instructions; Fig. 8(c) shows a sample plot depicting the behavior the HJ-FA kernels, for increasing HWTs, for the Mx-In input. For brevity, we omit the plots of the HJ-FAP kernels as we found their behavior to be similar. Compared to the X10 based kernels, these HJ kernels admit increased computational workload. This leads to a minor variation in their behavior compared to that of the plots shown in Fig. 8(a). For example, Fig. 8(a) shows a slight dip in performance in *HS* when we increase the HWTs from 16 to 32; such a dip is not visible in Fig. 8(c).

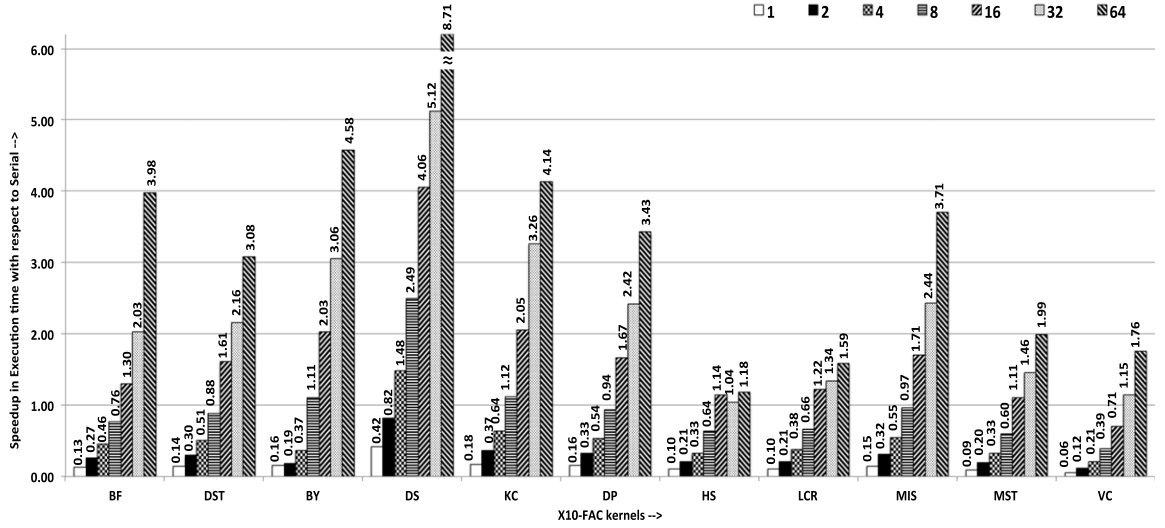
Recursive kernels: Fig. 9(a) and (b) depict the runtime characteristics of recursive X10-FA and X10-FAC kernels, respectively, with varying number of HWTs. It can be seen that the behavior displayed by these kernels is similar to their iterative counterparts. Similarly Fig. 10(a) and (b) depict the runtime characteristics of our recursive HJ-FA and HJ-FAP kernels, respectively with varying number of HWTs. Similar to their iterative counterparts, we use a workload of 10 million instructions for these recursive HJ kernels. It can be seen that performance for *MST* falls when the number of HWTs is 32 as compared to 16 HWTs. This is due to the relatively low value of workload in *MST*, which did not offset the increased contention among HWTs (compared to the case where the number of HWTs is set to 16) for the shared resources.

For brevity, in the rest of the section, we restrict ourselves to a subset of our benchmark kernels. We focus on the iterative HJ-FA and HJ-FAP kernels when the number of clusters is set to

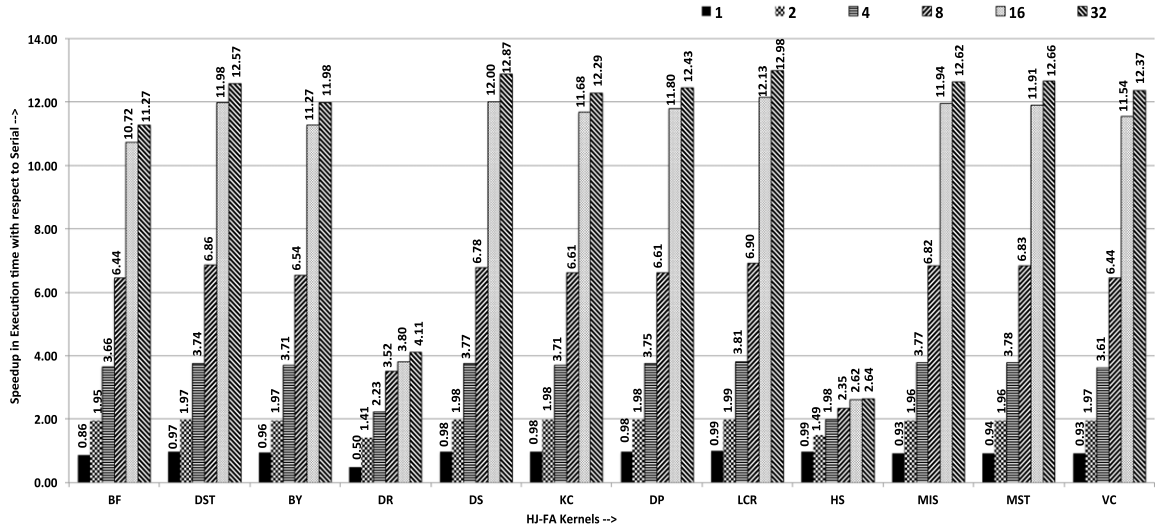
[5] We assume that the nodes are distributed equally among all the clusters.



(a) X10-FA plots; input size = 64 nodes, # clusters = 64.

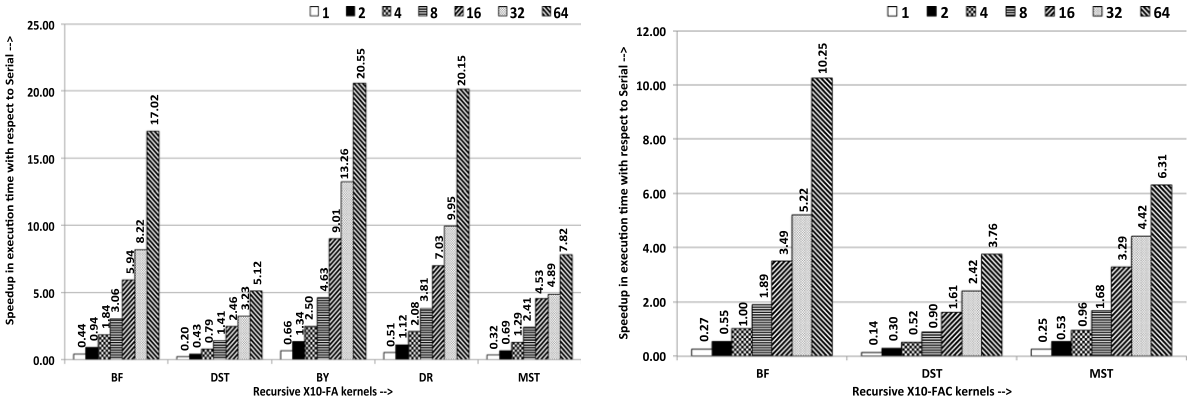


(b) X10-FAC plots; input size = 64 nodes, # clusters = 64.



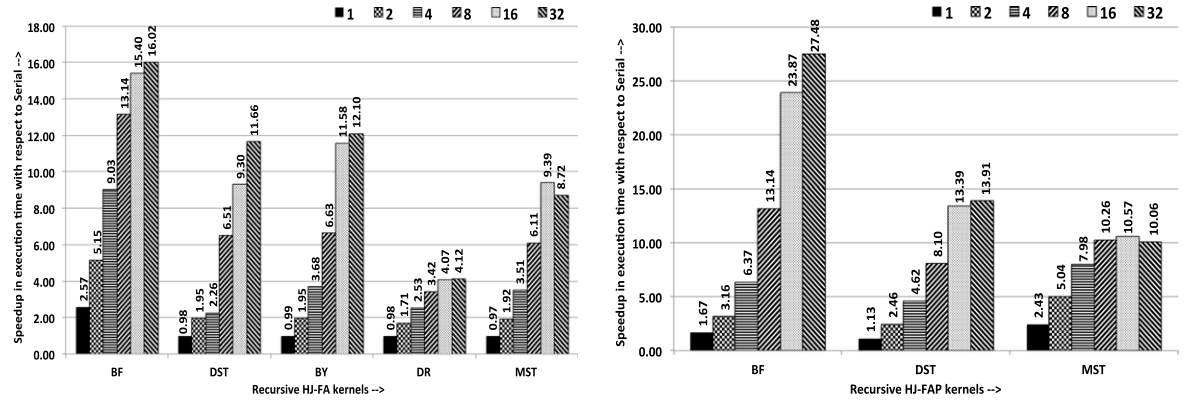
(c) HJ-FA plot; input size = 64 nodes, # clusters = 1, workload = 10 million instructions.

Fig. 8. Effect of varying HWTs on X10-FA, X10-FAC, and HJ-FA kernels.



(a) Recursive X10-FA plot. (b) Recursive X10-FAC plot.

Fig. 9. Plots for recursive kernels, for varying # HWTs; input size = 64 nodes, # clusters = 64.



(a) Recursive HJ-FA plot. (b) Recursive HJ-FAP plot.

Fig. 10. Plots for recursive kernels, for varying # HWTs; input size = 64 nodes, # clusters = 64.

1 (Section 6.2.4) and on the iterative X10-FA and X10-FAC kernels otherwise.

6.2.2. Effect of varying the input size (number of HWTs and number of clusters fixed)

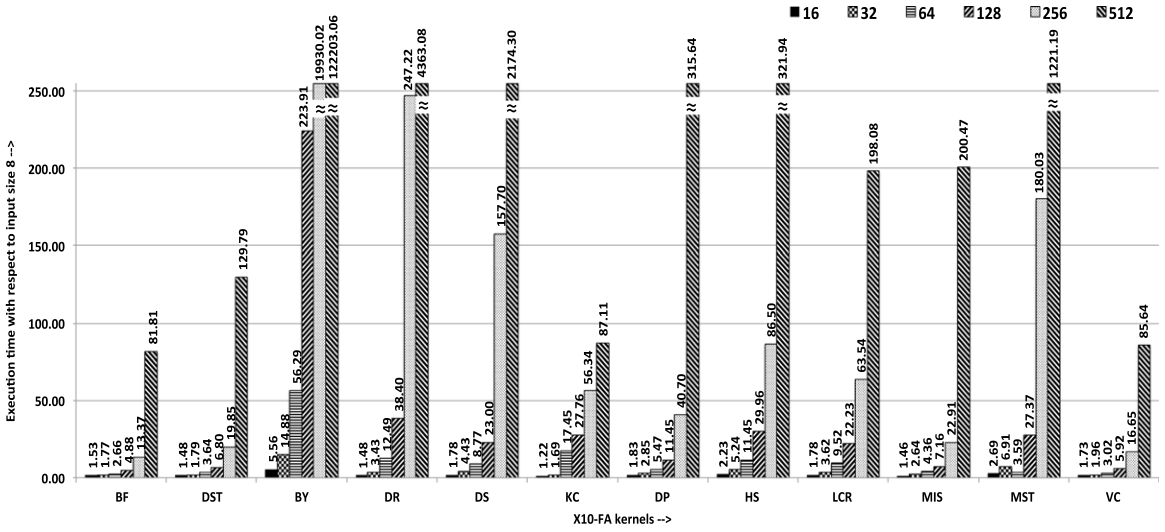
We vary the input size from 8 to 512 nodes, in multiples of two. For our simulations we set both the number of clusters and HWTs to eight. We choose eight HWTs for this study, as the communication between this set of HWTs does not involve any inter-processor or inter-hardware-node communication. For this study, the number of clusters could have been fixed at any one of 1, 2, 4, or 8 (if the number of clusters is more than eight then, for smaller inputs, some clusters may not contain any nodes). We broke the tie and set the number of clusters to 8; this leads to an interesting configuration where each cluster of nodes is simulated on a unique HWT and all the nodes in a cluster are simulated on a single HWT. We study the effect of varying the number of clusters in Section 6.2.3 and the combined effect of varying input size and number of clusters in Section 6.2.5.

Fig. 11 presents the behavior of X10-FA and X10-FAC kernels when run on the above-discussed configurations. It is evident from the plots that as the input size increases, the execution time for the kernels also increases. This behavior can be attributed to large inputs that lead to larger graphs (hence higher memory requirement), increased phases/rounds and communication. This effect is especially pronounced for *BY*, where the algorithm requires a large number of rounds to execute and in each round every node has to communicate with all the other nodes.

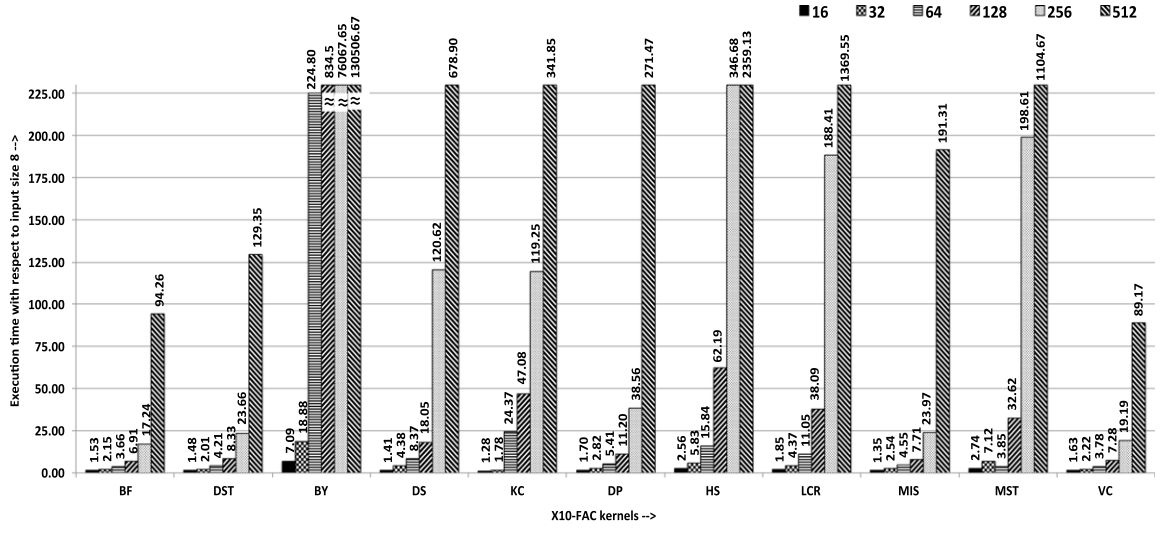
6.2.3. Effect of varying the number of clusters (input size and number of HWTs fixed)

We fix the HWTs to 8 (for the reasons mentioned above), and input size to 64 nodes. To study the effect of the clustering of nodes on the IMSuite kernels, we vary the number of runtime places from 1 to 64 (the upper limit is bound by the input size). Fig. 12 presents the behavior of X10-FA and X10-FAC kernels when run on the above-discussed configurations.

We note that (for the most part) as the number of places increases there is a proportional increase in the execution time of the benchmark kernels; this is owing to the increased communication cost between the tasks running on different places. Note that *BY* and *DR* in Fig. 12(a) and *VC*, *MST*, *LCR*, *KC*, *BY* and *DST* in Fig. 12(b) show a slight improvement in performance when the number of places is increased from 1 to 2. Based on our discussions with X10 developers, we conjecture that such curious behavior could be attested to gains resulting from a combination of inter-related factors concerning the distribution of tasks, such as change in cache access patterns and decreased contention in accessing the job queues (more places ⇒ more number of job queues for a given set of jobs ⇒ less contention for job selection). When we further increase the



(a) X10-FA.



(b) X10-FAC.

Fig. 11. X10-FA and X10-FAC plots for varying input size; # HWTs = # clusters = 8. The execution times are normalized with respect to the execution time when the input size is set to 8.

number of places the increased inter-place communication cost overshadows any such gains.

6.2.4. Effect of varying the input size and number of HWTs (number of clusters fixed)

We fix the number of clusters (places) to 1, to avoid the costs incurred due to inter-place communication. This setting also enables us to demonstrate the behavior of our HJ based kernels (HJ-FA and HJ-FAP); Fig. 13 shows the runtime characteristics of these kernels. Considering the lower/upper limits on the input size and HWTs, we vary these key parameters between 8 and 32, in sync. That is, when the input size is set to k, the number of HWTs is also set to k; we represent this configuration as Uk.

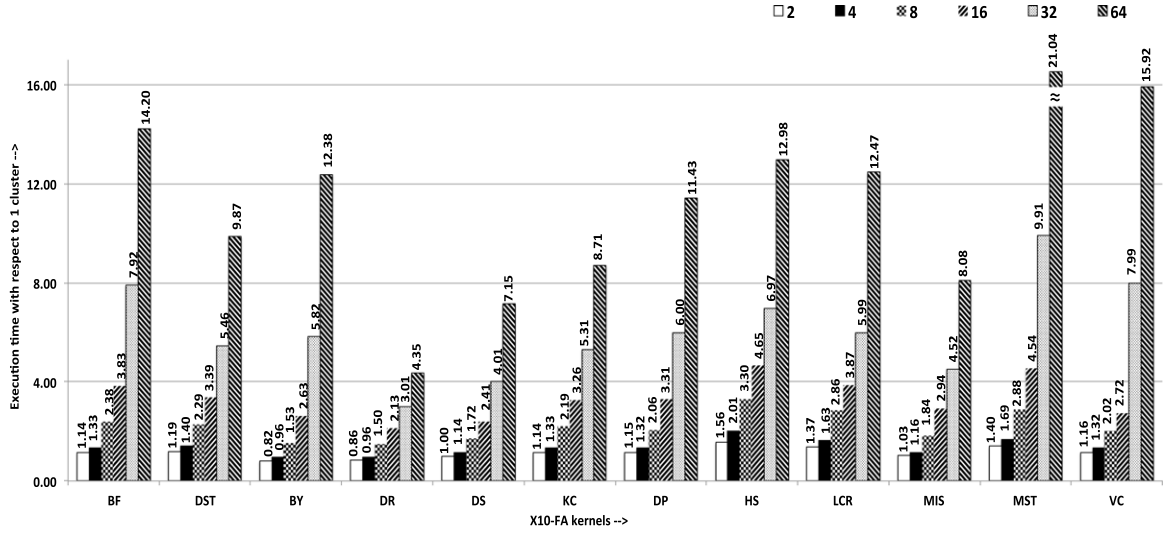
Note: one may naively assume that increasing the input size (say from 16 nodes to 32 nodes) will not lead to an increase in the execution time provided there is a proportional increase in the number of HWTs (say from 16 to 32). The behavior of our kernels shows that such a hypothesis does not hold. The execution time for all the kernel programs increases as we gradually move from U8 to U32. This is because of the increased computation and communication at each node, owing to the increased input. The exact quantum of increase depends on the working of the particular algorithm (amount of communication, computation and so on).

6.2.5. Effect of varying the input size and number of clusters (number of HWTs fixed)

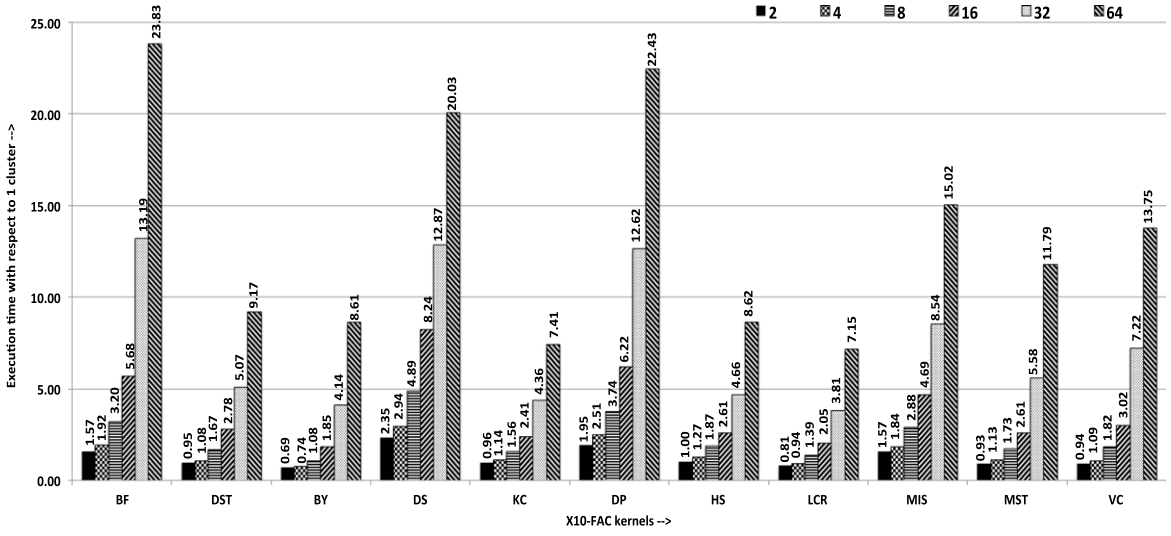
Fig. 14 displays the characteristics of X10-FA and X10-FAC kernels when the number of HWTs is set to 8 and the input size and number of clusters are varied from 8 to 128 (in multiples of two) in sync.[6] That is, when the input size is set to k, the number of clusters is also set to k; we represent this configuration as Vk.

Note that for higher configurations (such as V64 and V128) the execution times are significantly more than the lower ones (such as V8 and V16). This is because of three factors: (a) increase in input size leads to increased amount of work, (b) increase in number of

[6] While simulating the kernels for #clusters > 64, the X10 runtime often threw an error (ERRNO 104—connection reset by peer). To overcome this issue we performed all the runs (in plots that include larger clusters) by setting the environment variable X10_LAZYLINKS to true.



(a) X10-FA.



(b) X10-FAC.

Fig. 12. X10-FA and X10-FAC plots for varying # clusters; input size = 64, # HWTs = 8. The execution times are normalized with respect to the time taken when the # clusters is set to 1.

nodes leads to increase in overheads due to conflicts in accessing shared resources (such as hardware threads, memory and so on) by the tasks created, (c) increased number of clusters (places) leads to increased (inter-place) communication cost.

6.2.6. Effect of varying the number of clusters and number of HWTs (input size fixed)

This analysis helps us understand how the clustering and increased hardware threads affect the benchmarks. These characteristics have already been studied in Sections 6.2.1, 6.2.3 and 6.2.4. We avoid further analysis of the same, for brevity.

6.2.7. Effect of varying the input size, number of clusters and number of HWTs

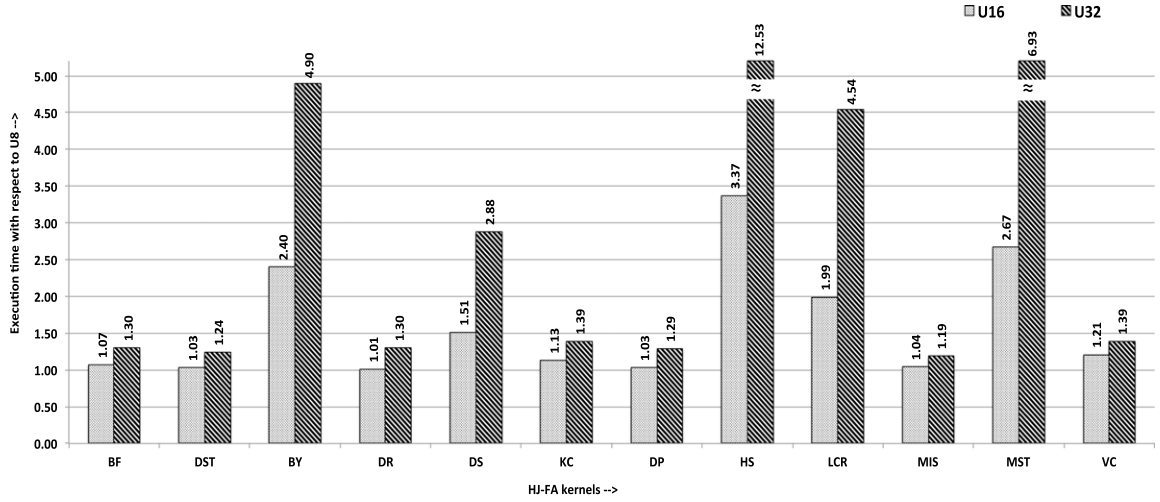
Fig. 15 shows the effect of varying all the three key parameters in sync: for running a kernel for input consisting of k nodes, we consider each node to be present in a unique cluster (thus, leading to k clusters), and the created tasks are run on k HWTs; we represent this runtime configuration as Wk.

In addition to the cost factors discussed in Section 6.2.4, we now incur an additional cost (resulting from inter-hardware-node communication) when we go from 32 → 64 → 128 HWTs. These cost factors explain the increase in program execution time as we go from W8 → W16 → W32 and a sharper increase as we go from W32 → W64 → W128. The exact quantum of increase depends on the working of the particular algorithm (amount of communication, computation and so on).

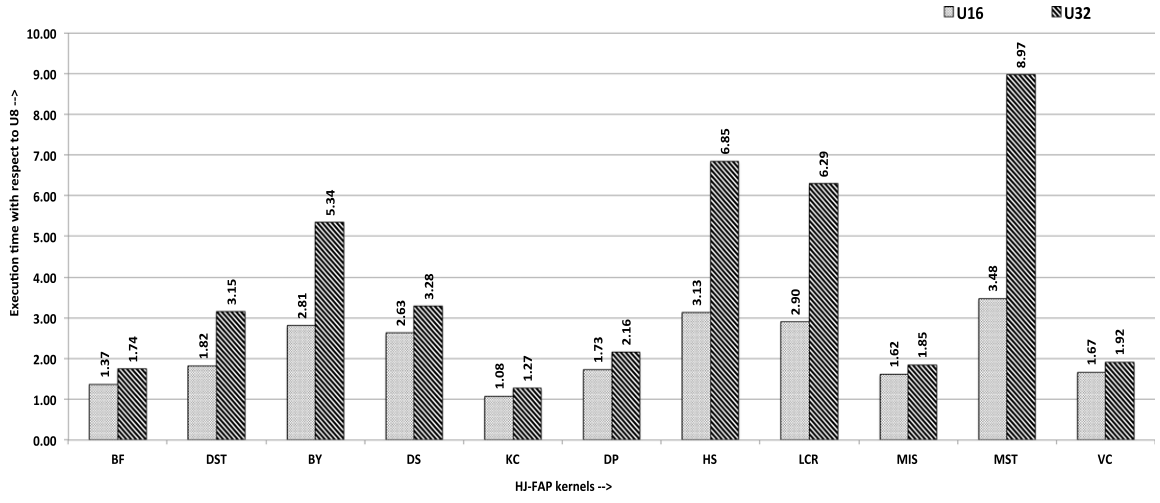
6.2.8. Effect of varying the input type

For the X10-FA and HJ-FA kernels, Fig. 16 plots the ratio of the execution time of these kernels for the input types Mx-In and Mn-In. We set the input size to 64 nodes and set the number of HWTs to 32 (the max number of HWTs that can be used by the HJ runtime on our hardware). The number of clusters (runtime places) is set to 64 for X10-FA kernels and 1 for the HJ-FA kernels.

It can be seen that besides the time taken for performing the underlying computation, the execution time of a benchmark is also dependent on (i) the number of task creation and termination



(a) HJ-FA.



(b) HJ-FAP.

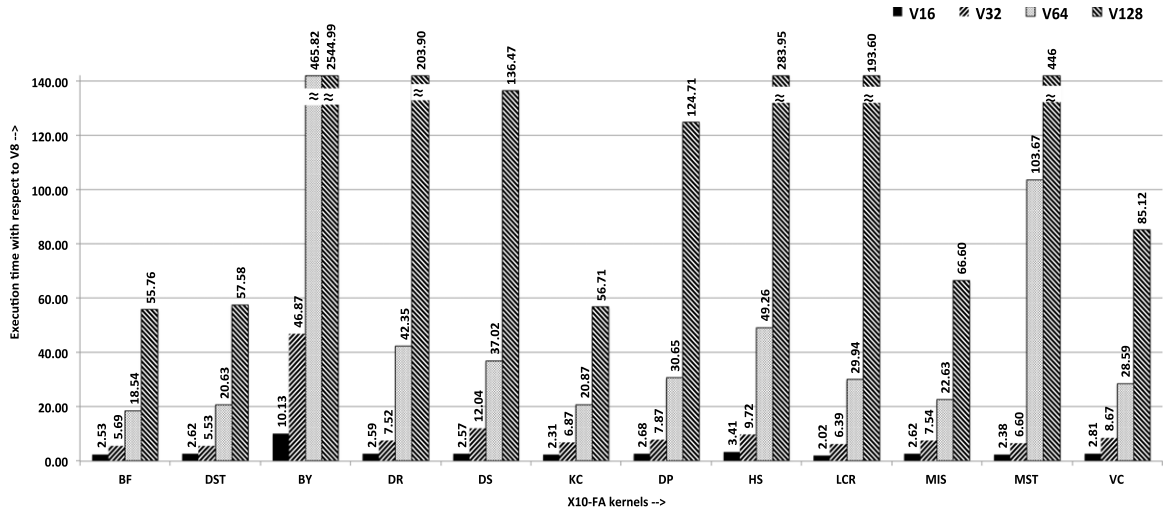
Fig. 13. HJ-FA and HJ-FAP plots for varying runtime configuration Un; n = # HWTs = input size; # clusters = 1. The execution time numbers are normalized with respect to that of U8.

operations, and (ii) the amount and cost of communication and mutex operations. For example, as shown in Fig. 7(a), in the *BY* kernel, the numbers of dynamic `finish` and `async` operations for Mn-In input (longer diameter D) are more than that of Mx-In (shorter diameter D); this increases the execution time for the HJ-FA version. However, in the context of X10-FA, the cost of decreased `finish` and `async` operations (in Mx-In) gets shadowed by the increased cost of communication, which is significantly higher than Mn-In; this leads to a reversal of behavior.

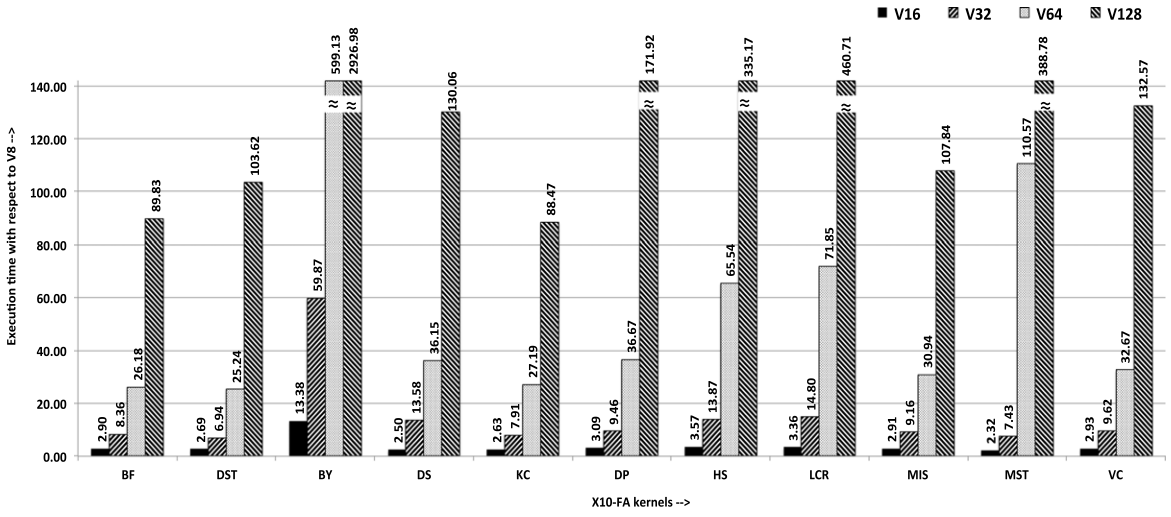
7. Scope of the benchmarks

We now briefly discuss the scope of the IMSuite kernels. We organize our discussion under four heads.

1. **Compiler optimizations and program analysis**: An optimizing compiler can use the IMSuite kernels to estimate its effectiveness in optimizing distributed applications (involving remote communication, barriers, load balancing and so on). The metrics advocated by the IMSuite kernels (such as number of task creation, join, mutex, barrier operations) and our kernel behavior evaluation schemes (distribution of communication; behavior with increase in the number of hardware threads, number of clusters and input size) can give meaningful insights for designing new optimizations. Further, the compiler writers can use these metrics and evaluation schemes to evaluate the overall effectiveness of their proposed optimizations. Developers of new parallel and distributed program analysis techniques can use the IMSuite kernels as the basis to test the effectiveness (scalability, precision and so on) of their proposed schemes.
2. **Runtime**: Developers of hypervisors, virtual machines and other managed runtimes can use the IMSuite to study and optimize the remote communications between different applications.
3. **Simulators**: Architecture and network simulators can use the communication trace generated by the IMSuite kernels, for varied inputs, to reason about the network traffic in the context of varied distributed systems.
4. **Study of distributed systems**: Though our analysis is shown in the context of a tightly coupled system, the IMSuite kernels can be run on both tightly and loosely coupled systems. This enables us to reason about different important aspects of distributed systems, even in the absence of expensive distributed hardware.



(a) X10-FA.



(b) X10-FAC.

Fig. 14. X10-FA and X10-FAC plots for varying runtime configuration Vn; n = # clusters = input size, # HWTs = 8. The execution time numbers are normalized with respect to that of V8.

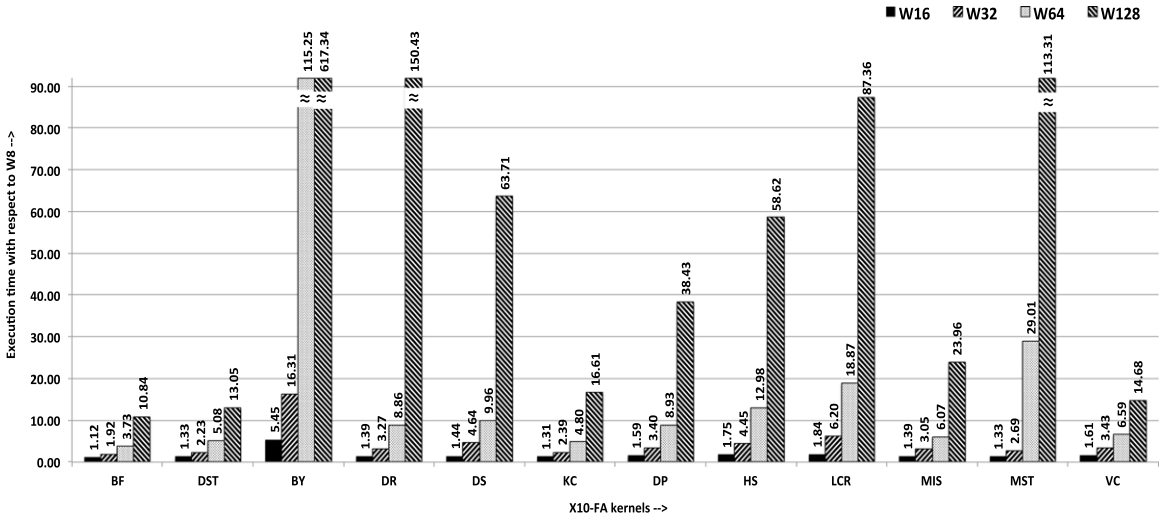
8. Conclusion

In this paper, we first identify a set of key requirements necessary for a kernel benchmark suite implementing distributed algorithms. We then present and characterize a new kernel benchmark suite (named IMSuite) that simulates twelve classical distributed algorithms (for varying input and runtime configurations) and meets all the key requirements. Currently, the kernels in IMSuite are available in two task parallel languages: X10 and HJ. Considering the different popular parallel programming styles, we present multiple variations of our kernels—31 parallel programs per language. To conveniently simulate varied configurations of distributed systems, we present an input generator and an output validator for each algorithm under consideration. IMSuite can be freely downloaded from http://www.cse.iitm.ac.in/~krishna/imsuite.

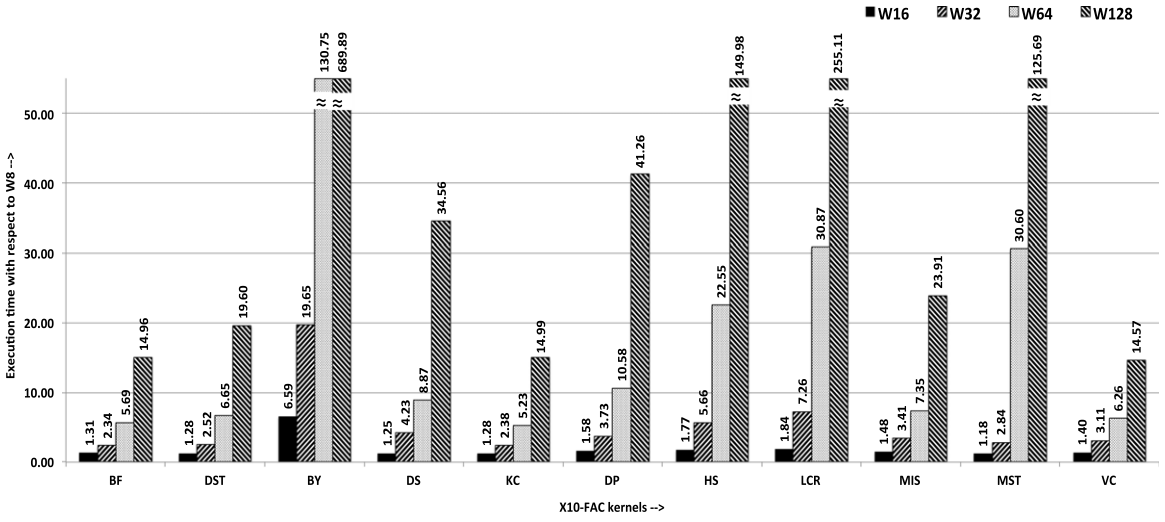
Future work: (A) We plan to expand the set of IMSuite kernels to cover more problem areas and algorithms in the domain of distributed computing. We also plan to expand IMSuite to cover other modern parallel and distributed languages, such as MPI, Chapel, Cilk, OpenMP, and COF. To help us in this direction, in near future, we plan to devise mechanisms to help the wider audience to contribute to IMSuite. (B) Thoroughly analyzing the behavior of IMSuite kernels by varying the backend (for example, Managed X10 vs. Native X10) and by varying the runtime scheduler (for example, Work-stealing vs. Work-sharing vs. Fork-Join) is another involved (yet, interesting) task left as future work.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions on an earlier version of this paper. We thank T V Kalyan, Tripti Warrier and John Jose for their comments on an earlier version of this paper. We acknowledge Pritam Majumder for the help in plotting the graphs and Aman Nougrahiya for the help in designing the Input Generators. We thank John Augustine and V. Kamakoti for the insightful discussions pertaining to popular distributed algorithms. We acknowledge the Virgo Supercomputing Cluster for the computing resources. We thank the X10 developers community, especially Igor Peshanksy and Vijay Saraswat, for their help in analyzing some of the obtained results (chiefly for the results in Section 6.2.3). This



(a) X10-FA.



(b) X10-FAC.

Fig. 15. X10-FA and X10-FAC plots for varying runtime configuration Wn; n = # HWTs = # clusters = input size. The execution time numbers are normalized with respect to W8.

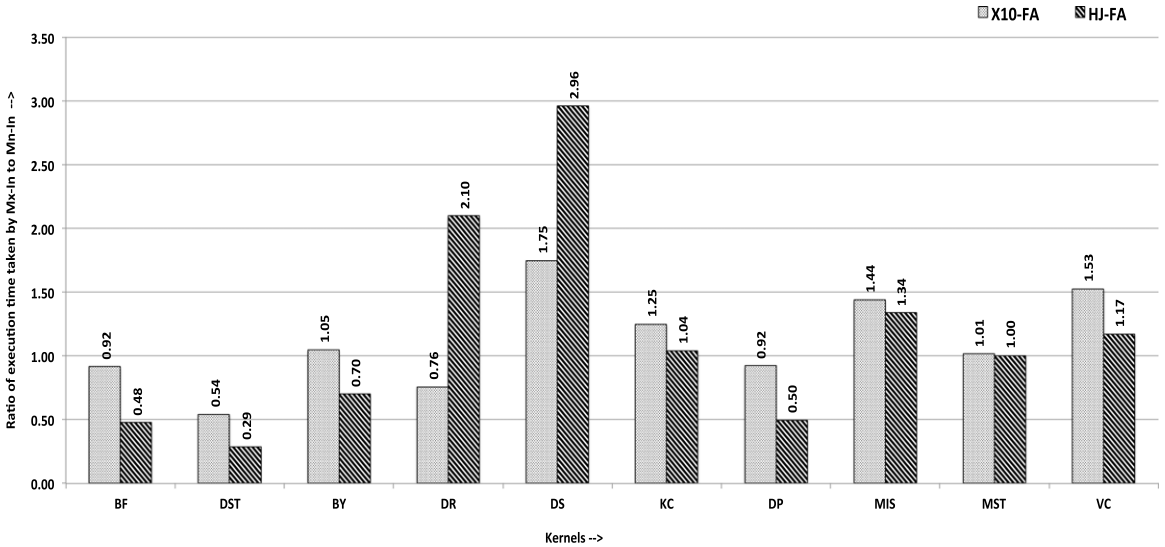


Fig. 16. Mx-In vs. Mn-In for X10-FA and HJ-FA; input size = 64, # HWTs = 32; # clusters = 1 (HJ-FA) and # clusters = 64 (X10-FA).

Name	X10-FA	X10-FAC	
	#Dynamic Fin	#Dynamic Fin	#Dynamic Bar
BF	$2 \times D$	D	D
DST	$(3 \times D^2 + 9 \times D)/2$	$D^2 + 3 \times D$	$\frac{D^2+3\times D}{2}$
BY	$(n/8 + 1)(2 \times D + 1)$	$(D + 1) \times (\frac{n}{8} + 1)$	$D \times (\frac{n}{8} + 1)$
DR	1	-	-
DS	$9 \times (\log^2 \Delta \times \log n)$	$\log^2 \Delta \times \log n$	$8 \times \log^2 \Delta \times \log n$
KC	$5 \times K^2$	$(2 \times K^2) + (3 \times K)$	$(3 \times K^2) + (3 \times K)$
DP	$8 \times D$	$2 \times D$	$6 \times D$
HS	$\log n \times (6 \times n + 1) + 1$	$1 + \log n \times (2 \times n + 1)$	$4 \times n \log n$
LCR	$2 \times n + 1$	$n + 1$	$n + 1$
MIS	$(3 \log_{4/3} m + 1) \times 4 + 1$	$1 + (1 + 3 \log_{4/3} m)$	$3 \times (1 + 3 \log_{4/3} m)$
MST	$\log n \times (3 \times D + 11)$	$(2 \times D + 6) \times \log n$	$(2 \times D + 7) \times \log n$
VC	$2 \log^* n + 9$	$\log^* n + 6$	$\log^* n + 3$

Fig. A.17. Dynamic characteristics of IMSuite Iterative kernels; D represents the diameter, n represents the number of nodes, Δ represents the maximum degree, and K represents the required maximum committee size.

Name	#Comm (FA)		#Mut		#Comm (FAC)	
	Mn-In	Mx-In	Mn-In	Mx-In	Mn-In	Mx-In
BF	1009	1145	126	766	568	956
DST	7500	3446	343	1348	5232	2816
BY	46386	97299	43K	95K	44874	96291
DR	20587	16204	0	0	–	–
DS	8020	31118	124	327	6193	27401
KC	5082	12504	2243	9666	4125	11556
DP	9996	12703	2625	8472	5649	10624
HS	12223	12223	0	0	8255	8255
LCR	25373	25373	0	0	9229	9229
MIS	1463	3204	1894	3518	1274	2637
MST	8890	19154	497	469	6055	13547
VC	1008	1208	0	0	756	890

Fig. A.18. X10-FA & X10-FAC dynamic communication and mutex operations; input size = 64 nodes.

research is partially supported by the New Faculty Seed Grant, funded by IIT Madras CSE/11-12/567/NFSC/NANV, DAE research grant CSE/13-14/139/BRNS/NANV and DST Fasttrack grant CSE/13-14/140/DSTX/NANV.

Appendix. Dynamic characteristics of IMSuite kernels

We discuss the dynamic characteristics of the iterative and recursive kernels separately. The terminology used (Mut, Bar, and Fin) is similar to that described in Section 6. We use the following additional abbreviation: #Comm represents the number of remote communications (excluding the barrier operations).

A.1. Iterative kernels

We present some of the dynamic characteristics of IMSuite iterative kernels in Fig. A.17. The number of dynamic finish and barrier statements vary as a function of the actual input. While the number of static async statements matches the number of static finish statements (as shown in Fig. 7(a)), the number of dynamic asyncs is n times the number of dynamic finish statements. Unlike the counts of the dynamic finish statements, the counts of the dynamic mutex and remote communications may depend on the structure of the input graph. Hence for these operations, we present the runtime characteristics (of programs written in X10-FA and X10-FAC) by comparing them for two specific inputs (Mx-In with 64 nodes and Mn-In with 64 nodes, both run on 64 runtime places), in Fig. A.18.

We avoid presenting the numbers for HJ-FA and HJ-FAP separately as the number of mutex operations match exactly that of X10-FA and X10-FAC, respectively, and since HJ-FA and HJ-FAP kernels run in the context of SP model they do not involve any remote communication.

Note that for a given input, the number of static (and dynamic) mutex operations is same for both X10-FA and X10-FAC kernels. This is because these two mainly differ in the synchronization primitives they use (see Section 5). For the same reason, the X10-FAC kernels have fewer number of static and dynamic finish (and async) operations compared to the X10-FA kernels.

A.1.1. Analysis of dynamic mutex operations and communication

As shown in Fig. A.18, the number of mutex operations for Mx-In is consistently higher than that of Mn-In, except in case of *MST*. The *MST* kernel has an interesting property that the number of mutex operations is guaranteed not to grow as we introduce some additional edges and corresponding unique weights. Thus the number of dynamic mutex operations for the Mx-In is not higher than that for Mn-In. The kernels *DR*, *HS*, *LCR*, and *VC* have no mutex operations and it is reflected in A.18.

An interesting point about *MIS* is that the amount of dynamic communication is less than the number of mutex operations. This is because in *MIS*, majority of remote communication operations involve mutex operation, but the other way round is not true.

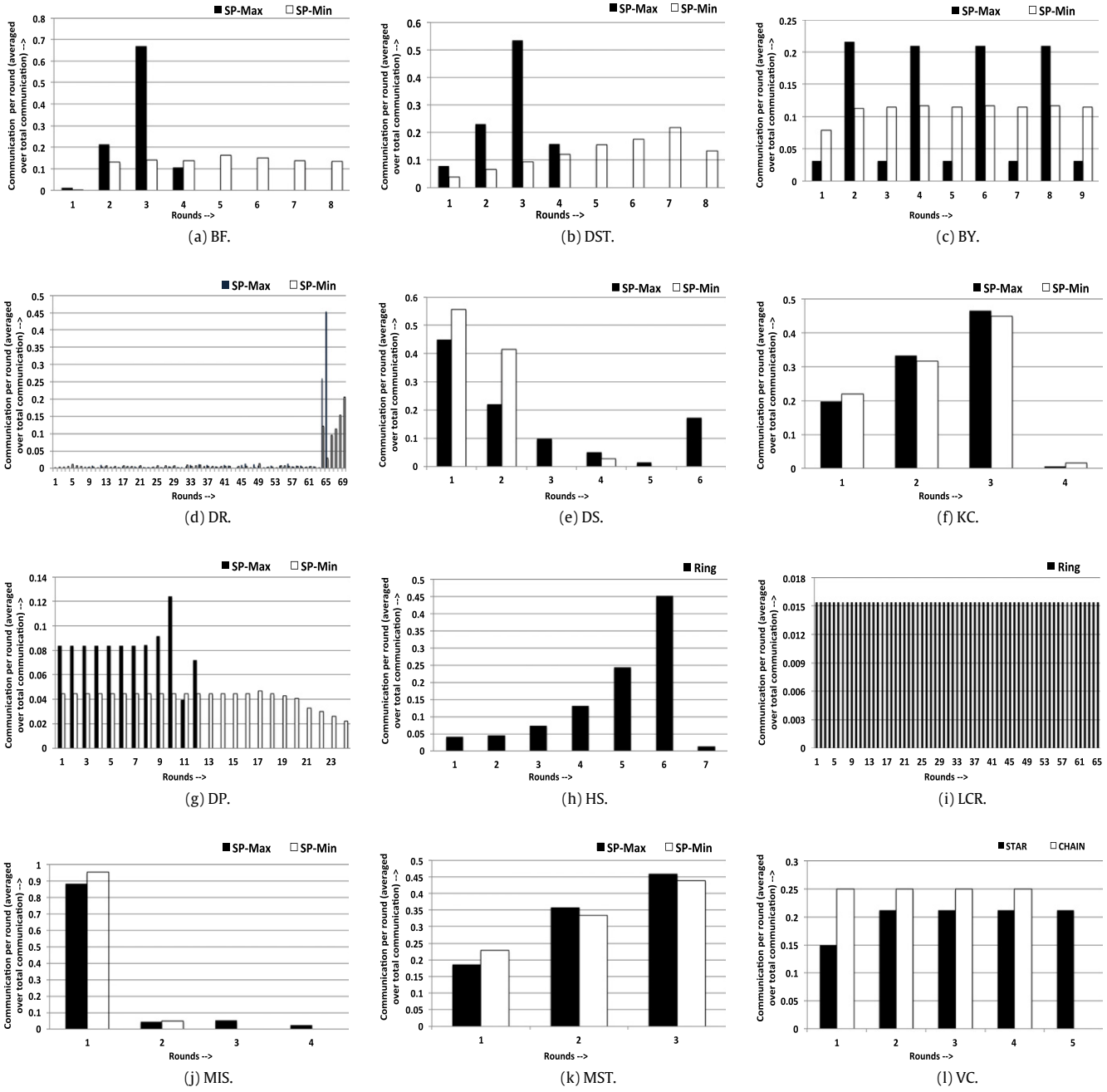
Note that, except for *DST* and *DR* kernels, rest all the kernels have higher amount of communication for Mx-In compared to Mn-In. One characteristic difference between Mx-In and Mn-In is that the latter increases the diameter of the graph and hence impacts the algorithms where an increase in diameter causes an increase in rounds. For *DST* and *DR* as the number of rounds increases, the number of messages (amount of remote communication) also increases.

A.1.2. Communication distribution

Fig. A.19 shows the amount of remote communication occurring in each round, for the X10-FAC kernels[7]; for the sake of illustration we set the input size to 64 nodes and present the results for two types of inputs: Mx-In and Mn-In. For *LCR* and *HS* we show only one curve as in the context of ring network Mx-In = Mn-In.

The behavior of *BY*, *KC* and *MST* for Mx-In and Mn-In is quite similar. Note that in *BY* for a specific input the amount of communication in each round is equal, but the communication per round in Mn-In is less than Mx-In. In case of *BF*, *DST*, *DR*, *DS* and *DP*, compared to Mn-In, the algorithm terminates in fewer rounds in case of Mx-In. However *MIS* and *VC* exhibit a contrasting behavior.

[7] Considering the case that we do not have a separate X10-FAC version for *DR*, we use the plot of the corresponding X10-FA version here.



(a) BF. (b) DST. (c) BY.

(d) DR. (e) DS. (f) KC.

(g) DP. (h) HS. (i) LCR.

(j) MIS. (k) MST. (l) VC.

Fig. A.19. X10-FAC plots for dynamic communication per round; input size = 64 nodes, # clusters = 64.

In *MIS*, compared to Mn-In where each node has fewer neighbors than Mx-In, in each round fewer nodes are added to the maximal-independent-set in case of Mx-In; thus increasing the number of rounds. In *VC*, as compared to Mn-In where the algorithm requires exactly one round to make the graph six colored, in Mx-In the number of rounds ($\geq$1) depends on the input. The shift-down operation (Section 3) on the other hand always takes three rounds to finish (irrespective of the input). For lack of space, we omit the communication distribution plots for the X10-FA and recursive kernels.

A.2. Recursive kernels

For our recursive kernels, Fig. A.20(a) and (b) present the runtime characteristics, for inputs Mn-In and Mx-In. For the most part, the comparative behavior (between Mn-In and Mx-In) displayed by these recursive kernels is similar to their iterative counterparts. A few points of interest: (i) the recursive *BY* kernel has more mutex operations than communication. This is because in the recursive kernel, majority of remote communication operations involve mutex operation, but the other way round is not true. (ii) in case of recursive *DST* kernel, the reduction in the amount of communication between the X10-FA and X10-FAC versions is directly impacted by the number of remote task creation operations present in the program: X10-FAC has comparatively fewer remote task creation operations than X10-FA. (iii) in case of *MST* the number of mutex operations (in X10-FA and X10-FAC) and barriers (in X10-FAC) are equal for both Mx-In and Mn-In inputs, as they are independent of the structure and type of input. However, the number of *async* and *finish* operations depends on the exact structure of the graph and the edge weights; thus making it hard to correlate the numbers for these two operations with the input types.

Name	#Async		#Fin		#Comm		#Mut	
	Mn-In	Mx-In	Mn-In	Mx-In	Mn-In	Mx-In	Mn-In	Mx-In
BF	252	2824	128	240	317	2945	317	2945
DST	2201	1092	256	151	2721	2569	189	1279
BY	73K	435K	36K	36K	74K	442K	109K	478K
DR	3476	3699	375	279	12469	12373	0	0
MST	1872	1870	258	345	15341	16460	255	255

(a) `X10-FA` recursive kernel characteristics; input size = 64 nodes.

Name	#Async		#Fin	
	Mn-In	Mx-In	Mn-In	Mx-In
BF	126	1615	64	140
DST	1689	836	248	147
MST	528	526	237	324

Name	#Comm		#Bar		#Mut	
	Mn-In	Mx-In	Mn-In	Mx-In	Mn-In	Mx-In
BF	317	3371	64	140	317	3371
DST	2217	2317	8	4	189	1279
MST	14026	15159	21	21	255	255

(b) `X10-FAC` recursive kernel characteristics; input size = 64 nodes.

Fig. A.20. Recursive kernels—runtime characteristics.

References

[1] R.J. Allison, S.P. Goodwin, R.J. Parker, S.F. Portegies Zwart, R. de Grijs, M.B.N. Kouwenhoven, Using the minimum spanning tree to trace mass segregation, Mon. Not. R. Astron. Soc. 395 (2009) 1449–1454.

[2] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, I.E. Venetis, A scalability benchmark suite for Erlang/OTP, in: Proceedings of the ACM SIGPLAN Workshop on Erlang, ACM, New York, NY, USA, 2012, pp. 33–42.

[3] V. Aslot, M.J. Domeika, R. Eigenmann, G. Gaertner, W.B. Jones, B. Parady, SPEComp: a new benchmark suite for measuring parallel computer performance, in: Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming, Springer-Verlag, London, UK, 2001, pp. 1–10.

[4] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, S.K. Weeratunga, The NAS parallel benchmarks-summary and preliminary results, in: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, SC'91, ACM, New York, NY, USA, 1991, pp. 158–165.

[5] C. Bienia, S. Kumar, J.P. Singh, K. Li, The PARSEC benchmark suite: characterization and architectural implications, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, ACM, New York, NY, USA, 2008, pp. 72–81.

[6] J.M. Bull, J. Enright, X. Guo, C. Maynard, F. Reid, Performance evaluation of mixed-mode OpenMP/MPI implementations, Int. J. Parallel Program. 38 (2010) 396–417.

[7] J.M. Bull, F. Reid, N. McDonnell, A microbenchmark suite for OpenMP tasks, in: Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 271–274.

[8] V. Cavé, J. Zhao, J. Shirako, V. Sarkar, Habanero-Java: the new adventures of old X10, in: Proceedings of the International Conference on Principles and Practice of Programming in Java, ACM, New York, NY, USA, 2011, pp. 51–61.

[9] P. Costa, M. Pasin, A.N. Bessani, M. Correia, Byzantine fault-tolerant MapReduce: faults are not just crashes, in: Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science, CLOUDCOM'11, IEEE Computer Society, 2011, pp. 32–39.

[10] C. Daly, J. Horgan, J. Power, J. Waldron, Platform independent dynamic Java virtual machine analysis: the Java grande forum benchmark suite, in: Proceedings of the Joint ACM-ISCOPE Conference on Java Grande, ACM, New York, NY, USA, 2001, pp. 106–115.

[11] M. Dayarathna, C. Houngkaew, T. Suzumura, Introducing ScaleGraph: an X10 library for billion scale graph analytics, in: Proceedings of the 2012 ACM SIGPLAN X10 Workshop, ACM, New York, NY, USA, 2012, pp. 6:1–6:9.

[12] M. Dayarathna, T. Suzumura, XGDBench: a benchmarking platform for graph stores in exascale clouds, in: Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), CLOUDCOM'12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 363–370.

[13] R.V. Der Wijngaart, M.A. Frumkin, NAS grid benchmarks version 1.0, NASA Technical Report NAS-02-005, NASA Ames Research Center, Morfett Field, CA, USA, 2002.

[14] A. Duran, X. Teruel, R. Ferrer, X. Martorell, E. Ayguade, Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP, in: Proceedings of the International Conference on Parallel Processing, IEEE Computer Society, Washington, DC, USA, 2009, pp. 124–131.

[15] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 2007, pp. 57–76.

[16] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: distributed graph-parallel computation on natural graphs, in: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 17–30.

[17] J. Gui, A. Liu, A new distributed topology control algorithm based on optimization of delay and energy in wireless networks, J. Parallel Distrib. Comput. 72 (8) (2012) 1032–1044.

[18] Habanero multicore software research project web page. https://wiki.rice.edu/confluence/display/HABANERO/HJ.

[19] D. Hasenkamp, A. Sim, M. Wehner, K. Wu, Finding tropical cyclones on a cloud computing cluster: using parallel virtualization for large-scale climate simulation analysis, in: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM'10, IEEE Computer Society, 2010, pp. 201–208.

[20] Intel MPI benchmarks: user guide and methodology description, October 2012.

[21] F. Jabeen, A.A.A. Fernandes, An algorithmic strategy for in-network distributed spatial analysis in wireless sensor networks, J. Parallel Distrib. Comput. 72 (12) (2012) 1628–1653.

[22] U. Kang, C.E. Tsourakakis, C. Faloutsos, PEGASUS: a peta-scale graph mining system implementation and observations, in: Proceedings of the 2009 Ninth IEEE International Conference on Data Mining, ICDM'09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 229–238.

[23] T. Kono, T. Fushiki, K. Asada, K. Nakano, Fuel consumption analysis and prediction model for "eco" route search, in: Proceedings of the 15th World Congress on Intelligent Transport Systems and ITS America's 2008 Annual Meeting, CLOUDCOM'11, 2008.

[24] F. Kuhn, R. Oshman, Dynamic networks: models and algorithms, SIGACT News 42 (1) (2011) 82–96.

[25] A. Lugowski, A. Alber, A. Buluç, J.R. Gilbert, S. Reinhardt, Y. Teng, A. Waranis, A flexible open-source toolbox for scalable complex graph analysis, 2012.

[26] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, D. Takahashi, The HPC challenge (HPCC) benchmark suite, in: Proceedings of the ACM/IEEE Conference on Supercomputing, ACM, New York, NY, USA, 2006.

[27] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers Inc., 1996.

[28] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD'10, ACM, New York, NY, USA, 2010, pp. 135–146.

[29] T. Milenković, V. Memišević, A. Bonato, N. Pržulj, Dominating biological networks, PLoS One 6 (8) (2011) e23016.

[30] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[31] M.S. Müller, M. Van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W.C. Brantley, C. Parrott, T. Elken, H. Feng, C. Ponder, SPEC MPI2007-an application benchmark suite for parallel systems using MPI, Concurr. Comput.: Pract. Exper. 22 (2010) 191–205.
[32] D. Peleg, Time-optimal leader election in general networks, J. Parallel Distrib. Comput. 8 (1990) 96–99.
[33] D. Peleg, Distributed Computing: A Locality-Sensitive Approach, Society for Industrial and Applied Mathematics, 2000.
[34] V. Saraswat, B. Bard, P. Igor, O. Tardieu, D. Grove, X10 language specification version 2.3, Tech. Rep., IBM, 2012.
[35] J. Shun, G.E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, SIGPLAN Not. 48 (8) (2013) 135–146.
[36] J. Shun, G.E. Blelloch, J.T. Fineman, P.B. Gibbons, A. Kyrola, H.V. Simhadri, K. Tangwongsan, Brief announcement: the problem based benchmark suite, in: Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA'12, ACM, New York, NY, USA, 2012, pp. 68–70.
[37] C. Siefert, E. Sturler, Probing methods for generalized saddle-point problems, Electron. Trans. Numer. Anal. 22 (2006) 163–183.
[38] J.P. Singh, W. Weber, A. Gupta, SPLASH: Stanford parallel applications for shared-memory, SIGARCH Comput. Archit. News 20 (1992) 5–44.
[39] E. Szèmeredi, Regular partitions of graphs, in: Problèmes Combinatoires et Thèorie des Graphes, Colloq. Internat. CNRS, 1976, pp. 399–401.
[40] A.S. Tanenbaum, Computer Networks, Pearson Education, India, 1985.
[41] I. Wallach, R. Lilien, The protein-small-molecule database, a non-redundant structural resource for the analysis of protein-ligand binding, Bioinformatics 25 (5) (2009) 615–620.
[42] R. Wattenhofer, Lecture Notes on Principles of Distributed Computing, Swiss Federal Institute of Technology, Zurich, 2011.
[43] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, SIGARCH Comput. Archit. News 23 (1995) 24–36.
[44] D.R. Zerbino, E. Birney, Velvet: algorithms for de novo short read assembly using de Bruijn graphs, Genome Res. 18 (2008) 821–829.

Suyash Gupta is currently pursuing M.S. in the department of Computer Science and Engineering at IIT Madras. He holds a B.Tech degree from Amity School of Engineering and Technology (affiliated to GGSIP University, New Delhi). He is interested in Compiler Optimizations and Multicore systems.

V. Krishna Nandivada is currently an Assistant Professor in the department of Computer Science and Engineering at IIT Madras. He has been associated with IBM India Research Lab, Sun Labs and Hewlett Packard for different periods of time. He holds a B.E. degree from Regional Engineering Colleges (now known as National Institute of Technology) Rourkela, M.E. degree from Indian Institute of Science, Bangalore, and a Ph.D. from University of California, Los Angeles. His research interests are Compilers, Program Analysis, Programming Languages, Fault Localization, and Multicore systems.