

# IMSuite: A Benchmark Suite for Simulating Distributed Algorithms

Suyash Gupta, V. Krishna Nandivada\*

*PACE Lab, Department of Computer Science and Engineering, IIT Madras, India 600036  
phone/fax: +91-44-22574380*

arXiv:1310.2814v1 [cs.DC] 10 Oct 2013

---

\*Corresponding author

*Email addresses:* [suyash@cse.iitm.ac.in](mailto:suyash@cse.iitm.ac.in) (Suyash Gupta), [nvk@cse.iitm.ac.in](mailto:nvk@cse.iitm.ac.in) (V. Krishna Nandivada)

*Preprint submitted to Elsevier*

*October 31, 2018*

---

**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.

*Keywords:* benchmarks, distributed algorithms, performance evaluation, task parallel, data parallel, recursive task parallel

---

## 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 [1, 2] and lecture notes [3] 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.
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 [4], NPB [5], BOTS [6], PARSEC [7].

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).

Problem	Applications and/or domains of interest.
Breadth first search	semantic graphs and community analysis.
Consensus	checking reliability of a system; grid computing; peer-to-peer networks; sensor networks.
Routing table	internet routing tables; OSPF protocol.
Dominating set	mobile adhoc network routing; mobile wireless adhoc networks.
Maximal independent set	symmetry breaking in networks; clustering in wireless ad hoc and sensor networks.
Committee creation	dynamic networks; token dissemination protocol.
Leader election	selecting a coordinating node for a network.
Spanning tree	in IEEE 1394.1 standard for interconnecting LANs using bridges.
Graph coloring	color growth bounded graphs such as unit disk graphs; for resolving resource conflicts.

Figure 1: Core distributed computing problems and their applications.

Our study of existing benchmark suites [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] 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.

Figure 1 shows some of the *core* problems in the area of distributed computing and lists a few of their 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 [1, 2, 3]. 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 [19] and HJ [20]. 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.

#### **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<sup>1</sup> (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 lan-

---

<sup>1</sup>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.

guage) 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 [5], BOTS [6], PARSEC [7] (including its two prior *avatars* SPLASH [17] and SPLASH-2 [18]), BenchERL [9], SPEC-OMP [10], JGF [12], NGB [13], SPEC-MPI [16] and HPCC [15]. While PARSEC, JGF, NPB and BenchERL also contain a few kernels, benchmark suites like PBBS [4] focus only on kernel benchmarks. Similarly, JGF contains a few micro-kernels as well, while EPCC [11, 8] and IntelMPI [14] 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 computation. 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

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

Figure 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  $\Delta$  denotes the maximum degree of the graph.

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 and non-scientific in nature. 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

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

**Breadth First Search (BFS):** We use two different BFS algorithms *BF* [3] and *DST* [3]. While *BF* outputs the distance of every node from the root, *DST* outputs the BFS tree.

**Byzantine Agreement:** The byzantine agreement (*BY*) algorithm [21] builds a consensus among the “good” nodes of a network that may also contain “faulty” nodes.

**Routing:** In the Dijkstra routing (*DR*) algorithm [22] each node in the network works independently and computes a routing table in parallel.

**Dominating Set:** The dominating set (*DS*) algorithm [3] creates a dominating set using

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

Figure 3: X10 command cheat sheet.

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 [3] partitions the input nodes into committees of size at most  $k$ .

**Maximal Independent Set**:  $MIS$  [3] 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 [1] 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 [23] works on a set of nodes organized in any general network.

**Minimum Spanning Tree**: The  $MST$  algorithm [3] 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 [3] 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.

### 3.2. X10 and HJ background

Figure 3 presents some constructs of X10 relevant to this manuscript (see the language manual [19] 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.

**Comparison with HJ:** The parallelism related constructs of Habanero Java [20] 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 [24] for details).

#### 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*. However, such an abstraction can pose a challenge in languages such as Java and OpenMP that neither support a notion like *places* nor can run on distributed memory systems. Programs in these languages can be seen as running all the tasks (corresponding to all the nodes) at a single place. This simulates a particular type of distributed system where all the data is “local” and the inter-node communication cost is minimal. X10 and HJ can mimic this behavior when they are restricted to run on a single place; we term it as the *Single-Place (SP) model*. 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 model, 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



```

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 ;

```

Figure 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; }

```

Figure 5: Structure of the abstract node for *LCR*

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. Figure 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 uni-directional 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 Figure 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  $j^{th}$  node can be set to  $j + 1$ ).

```

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;
          } } } } } }

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

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)

```

Figure 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.

Figure 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 distribution  $D$  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.

Figure 6(b) sketches the X10-FA kernel for *LCR*, showing only the differences with respect to the X10-FAC kernel shown in Figure 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 implementation 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: (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. 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 model of distribution 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 are 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<sup>2</sup> 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: *Star*, *Chain* and *Random*. The last one takes an additional input that specifies the maximum degree for any node. The choice

---

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

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, (ii) sparse graphs and (iii) random graphs (the edges are chosen at random). 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 kernels are derived from the algorithms that solve some of the core problems discussed in Figure 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 two hardware nodes<sup>3</sup>. Each hardware node of the cluster has two Intel E5-2670 2.6GHz processors, each processor has eight cores and each core can make use of (up to) two hardware threads. Thus, we can have up to 64 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. 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.

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:

---

<sup>3</sup>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.

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.

We use the results of the insightful paper of George et al [25] 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 and dynamic characteristics of our kernels. Figure 7(a) and Figure 7(b) present these characteristics for the iterative and recursive kernels, respectively, written in *X10-FA* and *HJ-FA*. Similarly, Figure 7(b) and Figure 7(c) present these characteristics for the recursive and iterative kernels, respectively, written in *X10-FAC* and *HJ-FAP*. In these tables, *Mut* is used as a generic name referring to mutex operations – *atomic* construct in *X10* and *isolated* construct in *HJ*. Similarly, *Bar* is used as a generic name referring to barrier operations – *Clock.advanceAll()* construct in *X10* and *next* in *HJ*. We use the following abbreviation: a) "#Static/Dynamic *Fin*" to represent number of static/dynamic *finish* operations, b) #Comm to represent the number of remote communications (excluding the barrier operations).

It can be seen that all the kernels in *IMSuite* are relatively small in size; their sizes vary approximately between 200 to 900 lines. Further the number of static *finish*, mutex and barrier statements are quite small. We discuss the characteristics of the iterative and recursive kernels separately.

#### 6.1.1. Iterative kernels

The number of dynamic *finish* and barrier statements vary as a function of the actual input. The number of static *async* statements matches the number of static *finish* statements, while 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 Figure 8.

Name	#Lines of Code		#Static		#Dynamic Fin
	X10-FA	HJ-FA	Fin	Mut	
<i>BF</i>	330	320	2	1	$2 \times D$
<i>DST</i>	510	500	7	3	$(3 \times D^2 + 9 \times D)/2$
<i>BY</i>	510	460	3	1	$(n/8 + 1)(2 \times D + 1)$
<i>DR</i>	370	355	1	0	1
<i>DS</i>	650	600	9	2	$9 \times (\log^2 \Delta \times \log n)$
<i>KC</i>	490	490	10	2	$5 \times K^2$
<i>DP</i>	440	395	4	1	$8 \times D$
<i>HS</i>	440	435	5	0	$\log n \times (6 \times n + 1) + 1$
<i>LCR</i>	260	245	3	0	$2 \times n + 1$
<i>MIS</i>	380	330	5	3	$(3 \log_{4/3} m + 1) \times 4 + 1$
<i>MST</i>	880	790	15	4	$\log n \times (3 \times D + 11)$
<i>VC</i>	420	395	5	0	$2 \log^* n + 9$

(a) X10-FA and HJ-FA iterative kernels.

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

(b) Recursive kernels.

Name	#Lines of Code		#Static			#Dynamic	
	X10-FAC	HJ-FAP	Fin	Bar	Mut	Fin	Bar
<i>BF</i>	330	310	1	1	1	$D$	$D$
<i>DST</i>	500	490	5	2	3	$D^2 + 3 \times D$	$\frac{D^2 + 3 \times D}{2}$
<i>BY</i>	505	455	2	1	1	$(D + 1) \times (\frac{n}{8} + 1)$	$D \times (\frac{n}{8} + 1)$
<i>DS</i>	580	510	1	8	2	$\log^2 \Delta \times \log n$	$8 \times \log^2 \Delta \times \log n$
<i>KC</i>	480	470	7	3	2	$2 \times K^2 + 3 \times K$	$3 \times K^2 + 3 \times K$
<i>DP</i>	430	375	1	3	1	$2 \times D$	$6 \times D$
<i>HS</i>	430	420	3	2	0	$1 + \log n \times (2 \times n + 1)$	$4 \times n \log n$
<i>LCR</i>	250	240	2	1	0	$n + 1$	$n + 1$
<i>MIS</i>	365	310	2	3	3	$1 + (1 + 3 \log_{4/3} m)$	$3 \times (1 + 3 \log_{4/3} m)$
<i>MST</i>	850	760	8	9	4	$(2 \times D + 6) \times \log n$	$(2 \times D + 7) \times \log n$
<i>VC</i>	410	390	3	2	0	$\log^* n + 6$	$\log^* n + 3$

(c) Iterative X10-FAC and HJ-FAP kernels.

Figure 7: Static characteristics of IMSuite kernels;  $D$  represents the diameter,  $n$  represents the number of nodes,  $\Delta$  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
<i>BF</i>	1009	1145	126	766	568	956
<i>DST</i>	7500	3446	343	1348	5232	2816
<i>BY</i>	46386	97299	43K	95K	44874	96291
<i>DR</i>	20587	16204	0	0	–	–
<i>DS</i>	8020	31118	124	327	6193	27401
<i>KC</i>	5082	12504	2243	9666	4125	11556
<i>DP</i>	9996	12703	2625	8472	5649	10624
<i>HS</i>	12223	12223	0	0	8255	8255
<i>LCR</i>	25373	25373	0	0	9229	9229
<i>MIS</i>	1463	3204	1894	3518	1274	2637
<i>MST</i>	8890	19154	497	469	6055	13547
<i>VC</i>	1008	1208	0	0	756	890

Figure 8: X10-FA & X10-FAC dynamic communication and mutex operations; input size = 64 nodes.

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.

**Analysis of dynamic mutex operations and communication:** As shown in Figure 8, 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 Figures 7(a), 7(c), and 8.

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.



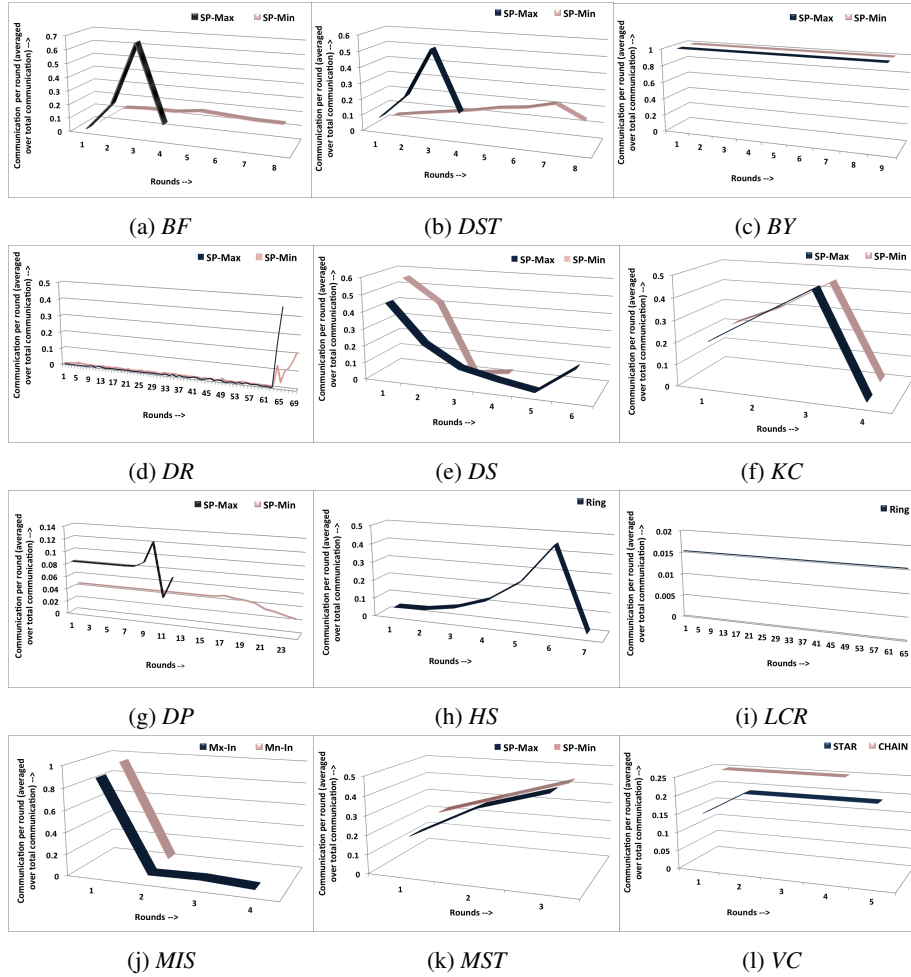


Figure 9: X10-FAC plots for dynamic communication per round; input size = 64 nodes, # clusters = 64.

**Communication distribution** Figure 9 shows the amount of remote communication occurring in each round, for the X10-FAC kernels<sup>4</sup>; 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 are 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 less than Mx-In. In case of *BF*, *DST*, *DR*, *DS* and

<sup>4</sup>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.

Name	#Async		#Fin		#Comm		#Mut	
	Mn-In	Mx-In	Mn-In	Mx-In	Mn-In	Mx-In	Mn-In	Mx-In
<i>BF</i>	252	2824	128	240	317	2945	317	2945
<i>DST</i>	2201	1092	256	151	2721	2569	189	1279
<i>BY</i>	73K	435K	36K	36K	74K	442K	109K	478K
<i>DR</i>	3476	3699	375	279	12469	12373	0	0
<i>MST</i>	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
<i>BF</i>	126	1615	64	140
<i>DST</i>	1689	836	248	147
<i>MST</i>	528	526	237	324

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

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

Figure 10: Recursive kernels - runtime characteristics.

*DP*, compared to Mn-In, the algorithm terminates in fewer rounds in case of Mx-In. However *MIS* and *VC* exhibit a contrasting behavior. 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.

### 6.1.2. Recursive kernels

For our recursive kernels, Figure 7(b) presents the static characteristics, and Figures 10(a) and 10(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 input, 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.

## 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<sup>5</sup>. 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 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 to 512 nodes. The execution times are too insignificant for inputs smaller than 8 nodes and it takes too long (of the order of several days) to execute programs with inputs larger than 512 nodes. We vary the number of HWTs between 1 to 64 (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 to 64; our version of X10 runtime throws a runtime error (ERRNO 104 - connection reset by peer) if we request more than 64 places on our machine.

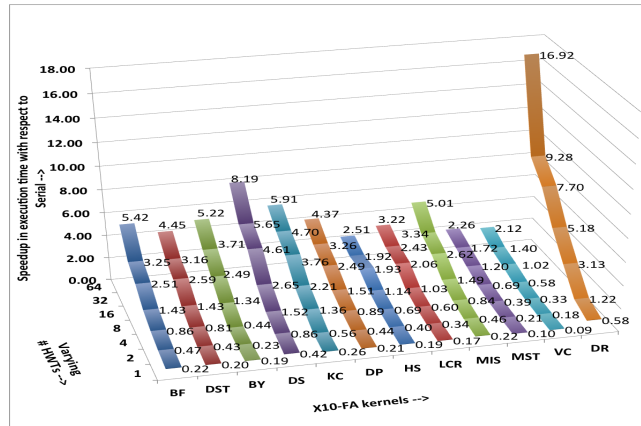
### 6.2.1. Effect of varying the number of HWTs (input size and number of clusters fixed)

Figure 11(a) and 11(b) present the execution time (speedup) statistics of the `X10-FA` and `X10-FAC` kernels, for varying number of hardware threads (1 to 64) in multiples of two. For all these runs we set the input variation to Mx-In and input size to 64 nodes. 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.

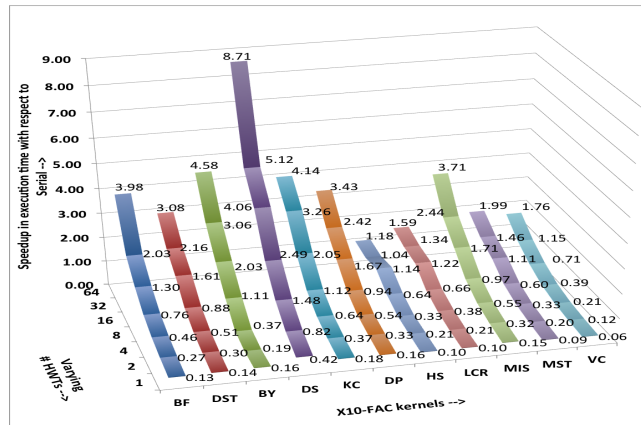
The performance improvement is more or less linear when we increase the number of hardware cores from 1 to 8 (intra-processor communication only) – this improvement is because of the increased sharing of workload among the HWTs.

---

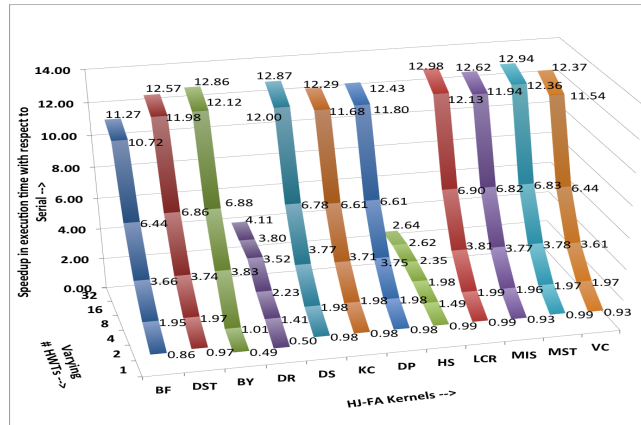
<sup>5</sup>We assume that the nodes are distributed equally among all the clusters.



(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.

Figure 11: Effect of varying HWTs on X10-FA, X10-FAC, and HJ-FA kernels.

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 Figure 11(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 runs 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; Figure 11(c) shows a sample plot depicting the behavior of 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 Figure 11(a). For example, Figure 11(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 Figure 11(c).

**Recursive kernels:** Figures 12(a) and 12(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 Figures 13(a) and 13(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 are 32 as compared to 16 HWTs. This is due to the relatively low value of workload in *MST*, which didn't 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 1 (Section 6.2.4) and on the iterative X10-FA and X10-FAC kernels otherwise.

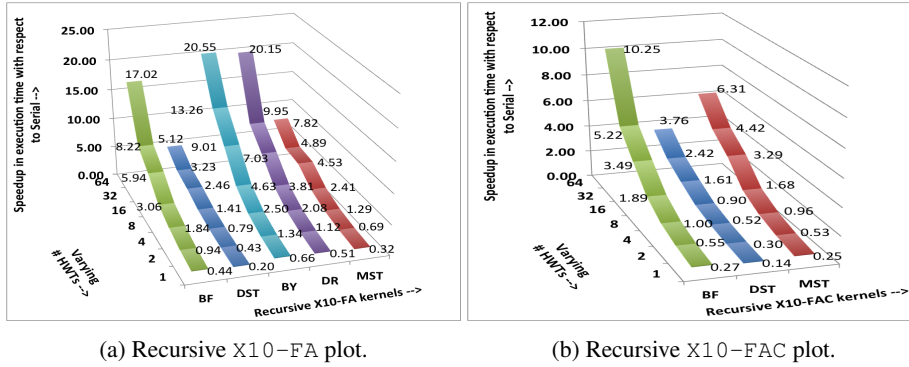


Figure 12: Recursive kernels plots for varying # HWTs; input size = 64 nodes, # clusters = 64.

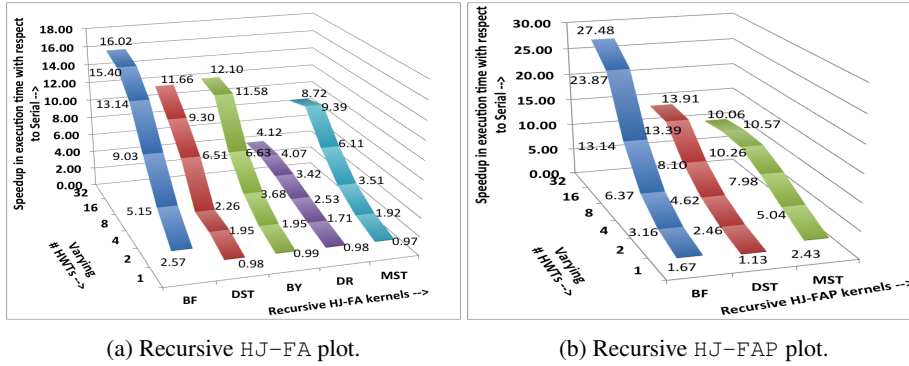
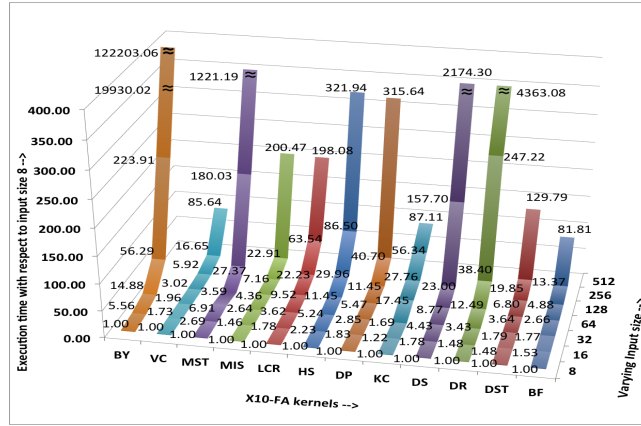


Figure 13: Recursive kernels plots for varying # HWTs; input size = 64 nodes, # clusters = 64.

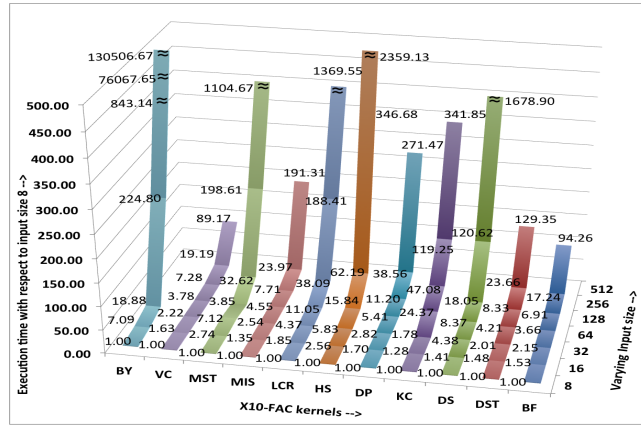
### 6.2.2. Effect of varying the input size (number of HWTs and number of clusters fixed)

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.

Figure 14 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),



(a) X10-FA



(b) X10-FAC

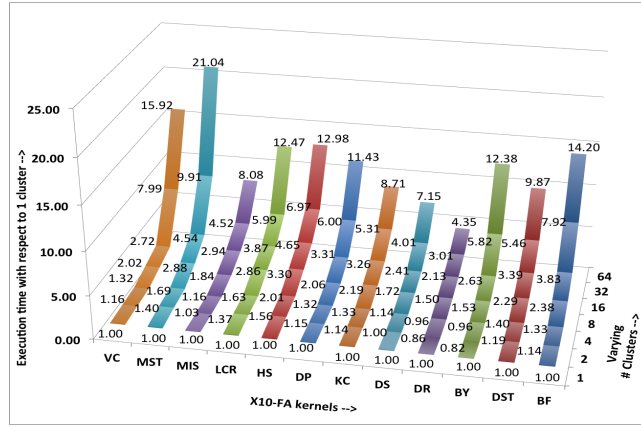
Figure 14: X10-FA and X10-FAC plots for varying input size; # HWTs = # clusters = 8. The execution times are normalized with respect to the execution time when the input size is set to 8.

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.

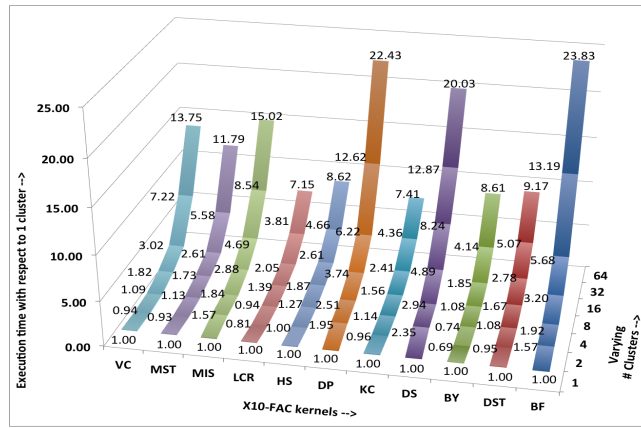
### 6.2.3. Effect of varying the number of clusters (input size and number of HWTs fixed)

To study the effect of the clustering of nodes on the IMSuite kernels, we vary the number of runtime places from 1 to 64. We fix the HWTs to 8 (for the reasons mentioned above), and input size to 64 nodes (to ensure that each cluster simulates at least one input node). Figure 15 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



(a) X10-FA

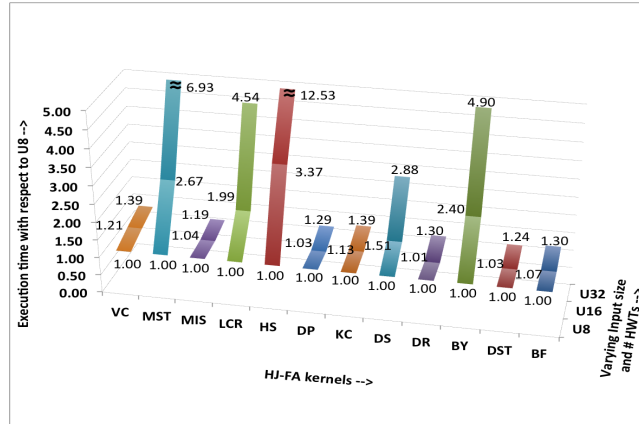


(b) X10-FAC

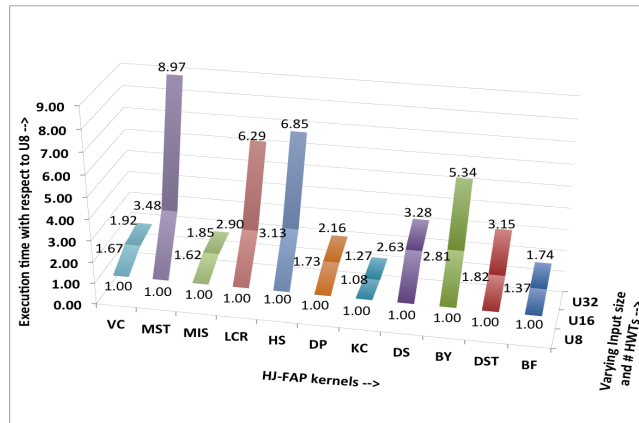
Figure 15: X10-FA and X10-FAC plots for varying # clusters; input size = 64, # HWTs = 8. The execution times are normalized with respect to the time taken when the # clusters is set to 1.

increase communication cost between the tasks running on different places. Note that *BY* and *DR* in Figure 15(a) and *VC*, *MST*, *LCR*, *KC*, *BY* and *DST* in Figure 15(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  $\Rightarrow$  more number of job queues for a given set of jobs  $\Rightarrow$  less contention for job selection). When we further increase the number of places the increased inter-place communication cost overshadows any such gains.





(a) HJ-FA



(b) HJ-FAP

Figure 16: HJ-FA and HJ-FAP plots for varying runtime configuration  $U_n$ ;  $n = \# \text{HWTs} = \text{input size}$ ;  $\# \text{clusters} = 1$ . The execution time numbers are normalized with respect to that of  $U_8$ .

#### 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-processor communication. This setting also enables us to demonstrate the behavior of our HJ based kernels (HJ-FA and HJ-FAP); Figure 16 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 to 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  $U_k$ .

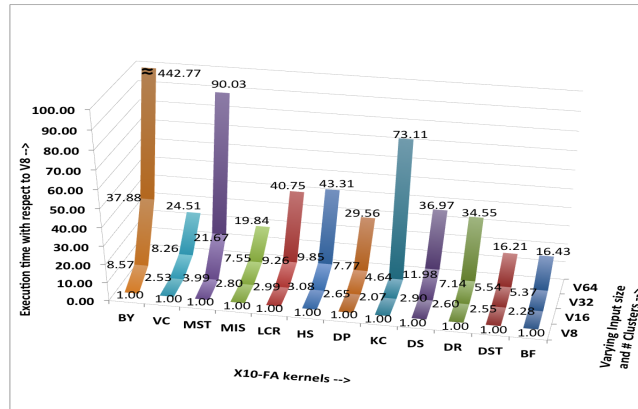
Note: one may naively assume that increasing the input size (say from 16 nodes to 32 nodes) will not lead to an increase in the execution time provided there is a proportional increase in the number of HWTs (say from 16 to 32). The behavior of our kernels show that such a hypothesis does not hold. The execution time for all the kernel programs increases as we gradually move from  $U_8$  to  $U_{32}$ . This is because of

the increased computation and communication at each node, owing to the increased input. Further, the rate of increase in execution time is less when we move from U8 to U16, compared to the case when we move from U16 to U32. This is because of the increased resource conflicts between the hardware threads, in the later case. The exact quantum of increase depends on the working of the particular algorithm (amount of communication, computation and so on).

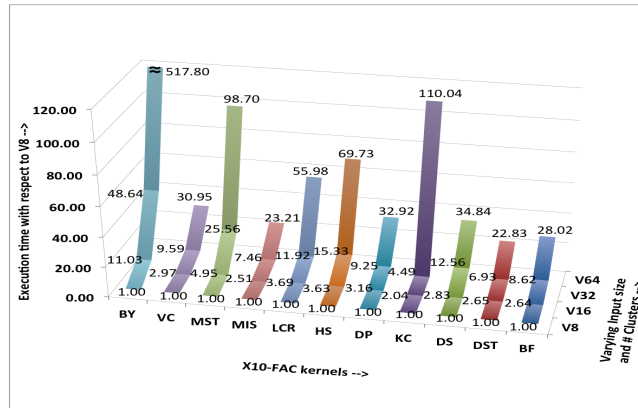
6.2.5. *Effect of varying the input size and number of clusters (number of HWTs fixed)*

Figure 17 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 64 (in multiples of two) in sync. 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 the execution time of higher configurations (such as V64) is significantly



(a) X10-FA



(b) X10-FAC

Figure 17: 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.

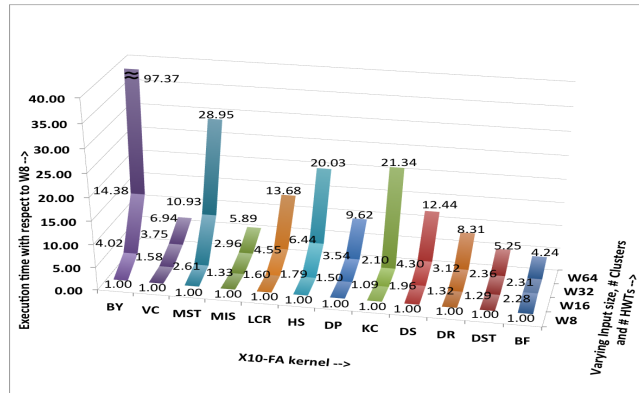
more than the lower ones (such as V8). This is because of three factors: a) increase in input size leads to increased amount of work, b) increase in number of nodes leads to increase in overheads due to conflicts in accessing shared resources (such as hardware threads, memory and so on) by the tasks 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)*

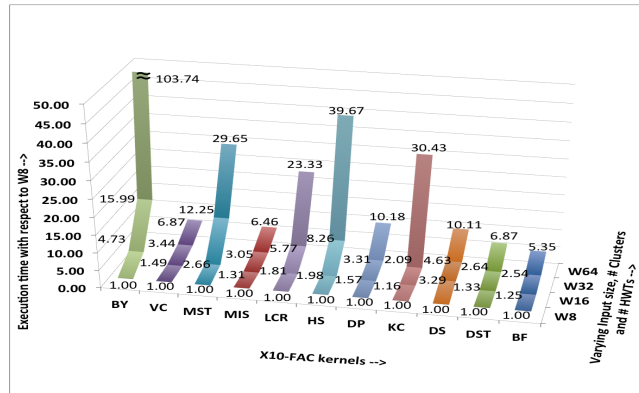
This analysis helps understand how the clustering and increased hardware threads affect the benchmarks. These characteristics have already been studied in Sections 6.2.1, 6.2.3 and 6.2.4. We avoid further analysis of the same, for brevity.

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

Figure 18 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



(a) X10-FA



(b) X10-FAC

Figure 18: 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.

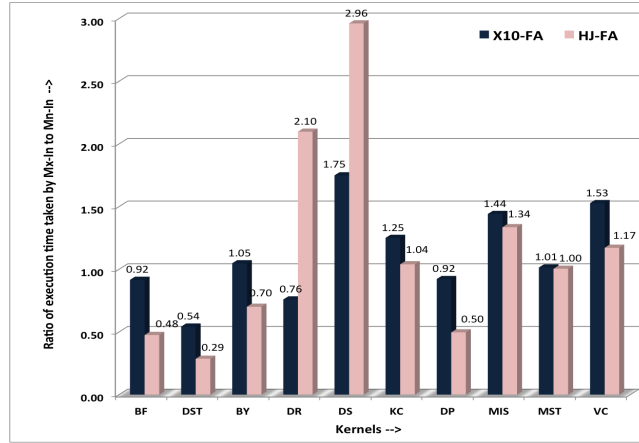


Figure 19: Mx-In Vs Mn-In for X10-FA and HJ-FA; input size = 64, # HWTs = 32; # clusters = 1 (HJ-FA) and # clusters = 64 (X10-FA)

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 to 64 HWTs. These cost factors explain the increase in program execution time as we go from W8 to W32 and a sharper increase as we go from W32 to W64. 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, Figure 19 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 operations, and (ii) the amount and cost of communication and mutex operations. For example, as shown in Figure 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.

## 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>.

Our proposed benchmarks can be ported to other languages that support distributed memory (such as `MPI`) and shared memory (such as `OpenMP`). We believe that our evaluation of `IMSuite` in the context `UP` and `SP` models of distribution, gives us some indication of how the future `MPI` and `OpenMP` ports of `IMSuite` may behave.

## Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions on an earlier version of this paper. We thank T V Kalyan, Tripti Warriar and John Jose for their comments on an earlier version of this paper. We thank the X10 developers community, especially Igor Peshankysy and Vijay Saraswat, for their help in analysing some of the obtained results (chiefly for the results in Section 6.2.3). This 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.

## References

- [1] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers Inc., 1996.
- [2] D. Peleg, *Distributed computing: A Locality-Sensitive Approach*, Society for Industrial and Applied Mathematics, 2000.
- [3] R. Wattenhofer, *Lecture notes on Principles of Distributed Computing*, Swiss Federal Institute of Technology Zurich, 2011.
- [4] 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: *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, ACM, New York, NY, USA, 2012, pp. 68–70.
- [5] 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. Venkatakrisnan, S. K. Weeratunga, The NAS Parallel Benchmarks-Summary and Preliminary Results, in: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, ACM, New York, NY, USA, 1991, pp. 158–165.
- [6] 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.
- [7] 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.
- [8] J. M. Bull, J. Enright, X. Guo, C. Maynard, F. Reid, Performance Evaluation of Mixed-Mode OpenMP/MPI Implementations, *International journal of parallel programming* 38 (2010) 396–417.

- [9] 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.
- [10] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, B. Parady, SPECComp: 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.
- [11] 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, IWOMP'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 271–274.
- [12] 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.
- [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] Intel MPI Benchmarks: User Guide and Methodology Description (October 2012).
- [15] 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.
- [16] 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.
- [17] J. P. Singh, W. Weber, A. Gupta, SPLASH: Stanford Parallel Applications for Shared-Memory, *SIGARCH Comput. Archit. News* 20 (1992) 5–44.
- [18] 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.
- [19] V. Saraswat, B. Bard, P. Igor, O. Tardieu, D. Grove, X10 Language Specification Version 2.3, Tech. rep., IBM (2012).
- [20] 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.

- [21] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
- [22] A. S. Tanenbaum, Computer Networks, Pearson Education India, 1985.
- [23] D. Peleg, Time-Optimal Leader Election in General Networks, J. Parallel Distrib. Comput. 8 (1990) 96–99.
- [24] Habanero Multicore Software Research Project web page, <https://wiki.rice.edu/confluence/display/HABANERO/HJ>.
- [25] 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.