X10 for Productivity and Performance at Scale

Olivier Tardieu, David Grove, Benjamin Herta,
Vijay Saraswat, Mikio Takeuchi, Wei Zhang
IBM Watson & TRL

Tomio Kamada
Kobe U/RIKEN

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002, by the Department of Energy, and by the Air Force Office of Scientific Research.

Some of the results were obtained by using the K computer at the RIKEN Advanced Institute for Computational Science. This research was partially supported by JST, CREST.

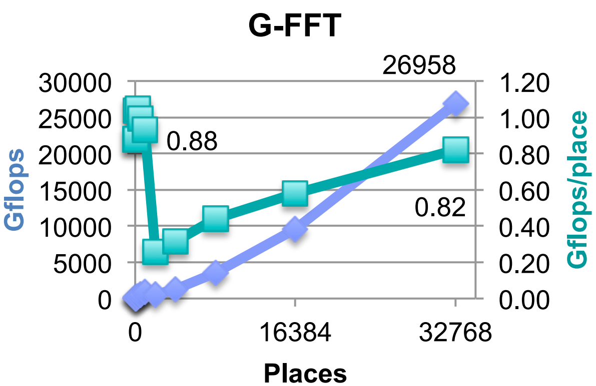
This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

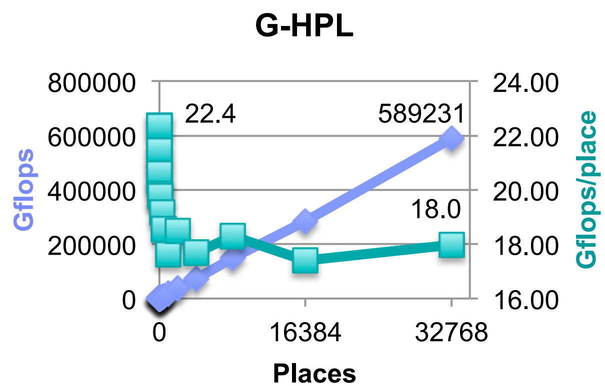
X10

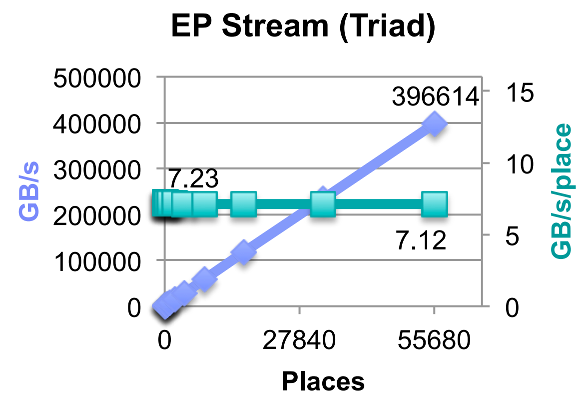
X10 is

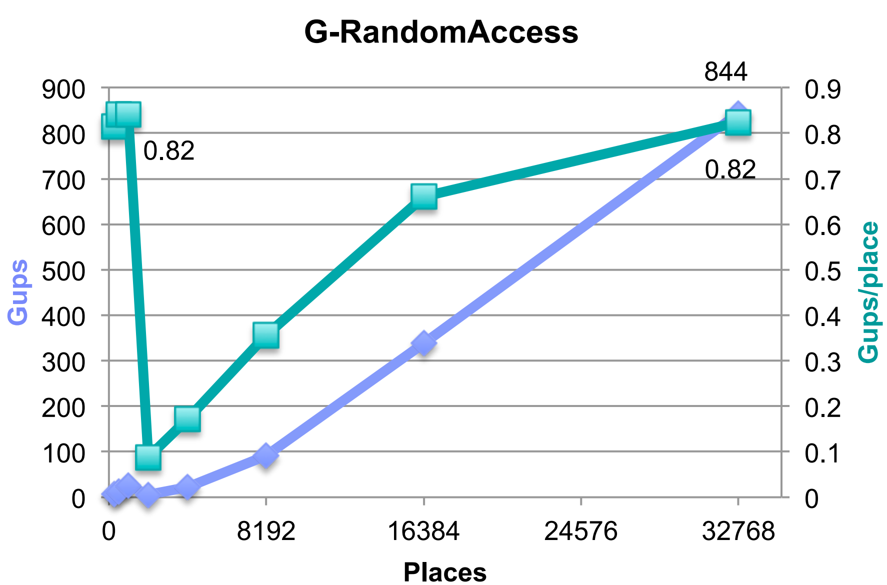
- A programming language
  - evolution of Java
    - object-oriented, imperative, strongly typed, garbage collected
  - focus on *scale*
    - HPC and Big Data
  - focus on *productivity*
- An implementation of the APGAS programming model
  - Asynchronous Partitioned Global Address Space
    - PGAS: single address space but with internal structure (➔ locality control)
    - asynchronous: task-based parallelism, active-message-based distribution
- A tool chain
  - compiler, runtime, standard library, IDE
  - open-source *research* prototype
  - portable and interoperable

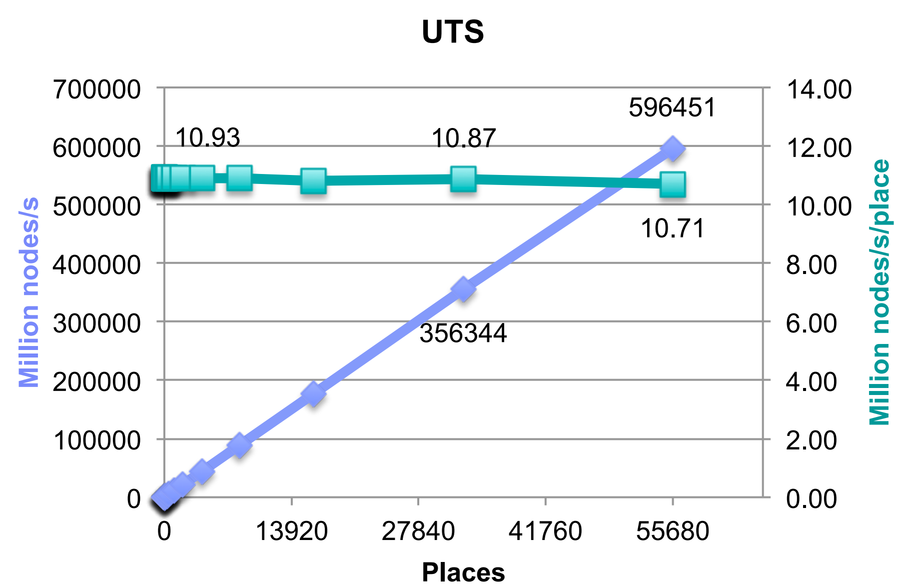
2012 HPC Challenge Class 2 – Best Performance Award











2013 HPC Challenge Class 2 Highlights

- 3 + 2 kernels
  - 3 classic kernels: EP Stream (Triad), Global HPL, Global FFT
  - 2 irregular kernels: Unbalanced Tree Search (UTS), Betweenness Centrality (BC)

- With a twist…
  - direct *and* library-based implementations of UTS and BC
    - UTS with our 2012 ad hoc load balancer, BC with randomized load partitioning
    - UTS and BC using a common Global Load Balancing Library (GLB)

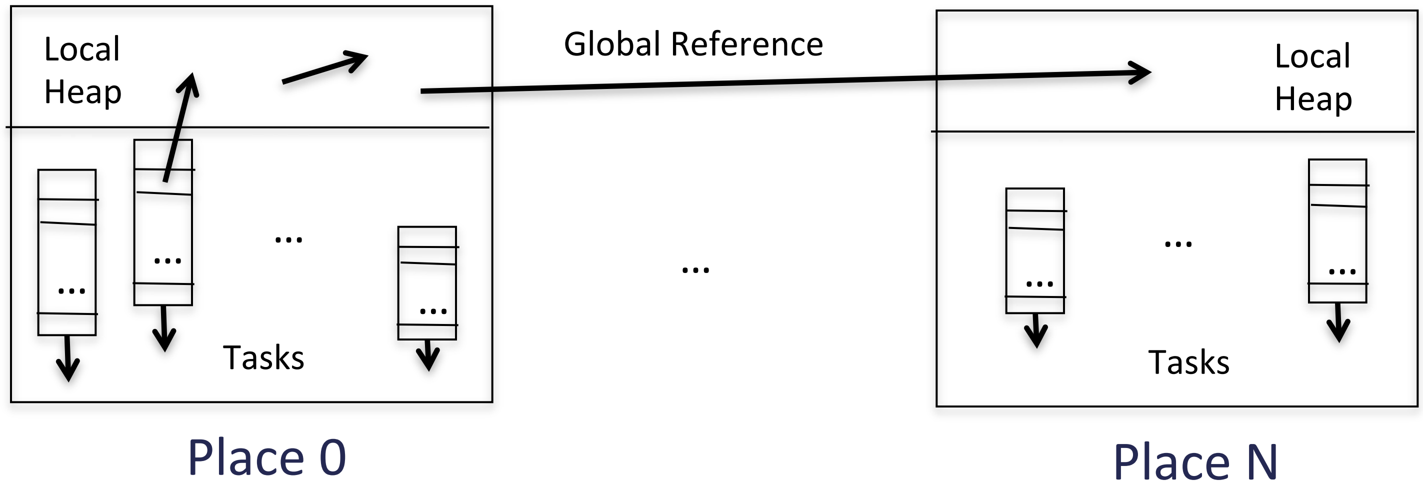
- On 3 different architectures
  - Power 775 (256 cores), BlueGene/Q (16384 cores), K Computer (8192 cores)
  - using a new MPI transport backend for X10

- With major productivity improvements
  - 64-bit arrays, native code snippets…

Outline

- X10/APGAS overview
- Regular kernels
  - productivity
  - performance
- Irregular kernels
  - motivation
  - productivity
  - performance

APGAS Concepts: Places and Tasks



Task parallelism

- **async** S
- **finish** S

Place-shifting operations

- **at**(p) S
- **at**(p) e

Concurrency control

- **when**(c) S
- **atomic** S

Distributed heap

- **GlobalRef**[T]
- **PlaceLocalHandle**[T]

APGAS Idioms

- Remote procedure call

```
v = at(p) evalThere(arg1, arg2);
```

- Active message

```
at(p) async runThere(arg1, arg2);
```

- Divide-and-conquer parallelism

```
def fib(n:Int):Int {
  if(n < 2) return n;
  val f1:Int;
  val f2:Int;
  finish {
    async f1 = fib(n-1);
    f2 = fib(n-2);
  }
  return f1 + f2;
}
```

- SPMD

```
finish for(p in PlaceGroup.WORLD) {
  at(p) async runEverywhere();
}
```

- Atomic remote update

```
at(ref) async atomic ref() += v;
```

- Computation/communication overlap

```
val acc = new Accumulator();
while(cond) {
  finish {
    val v = acc.currentValue();
    at(ref) async ref() = v;
    acc.updateValue();
  }
}
```

IBM

Productivity: Portability

- X10 compiles to Java or C++
- X10 runs on AIX, Linux, OS X, Windows

- X10 historically ran on
  - shared memory transport: not scalable
  - TCP/IP transport: portable but not high-performance and only for medium scale
  - PAMI transport: high-performance but not portable

- New in 2013: X10 runs on MPI transport (next release)
  - runs across a variety of hardware including Power 775, BlueGene/Q and K
  - supports full X10
    - arbitrary mix of active messages and collectives
  - only requires MPI 2
    - we prefix blocking collective calls with our own non-blocking barriers
    - we will use MPI 3 non-blocking barriers instead if available

HPL

```
def panelFactorization(...) {
  ...
  if (ownerOf(I, I)) {
    // we own the diagonal element -> master for now
    ...
    // find row J with largest value on column I
    val J = column.indexOfMax(...); // collective
    ...
    // swap row I with row J
    rowExchange(I, J, ...); // naturally one-sided -> active message
    ...
  }
  ...
  column.broadcast(...); // collective
  // must be non-blocking as master may request row exchange from slave
  ...
}
```

Productivity: Arrays

- New in 2013: redesigned X10 arrays for X10 2.4.0
  - 64-bit indexing – Long is now the default integer type
  - focus on performance *and* ease of use
    - 0-based, dense, rectangular arrays
    - 1D, 2D, and 3D row-major arrays
    - support for arrays of complex numbers
  - multi-dimensional arrays are fully implemented in X10
    - on top of primitive 0-based, dense, 1D arrays

- You no longer need a PhD to use X10 arrays
  - FORTRAN style & performance

- It is trivial to customize arrays to your needs or define your own
  - column-major, 1-based, 6D, etc…
  - using copy & paste

FFT before X10 2.4.0

```
val A:IndexedMemoryChunk[Double]; // 2D -> 1D, Complex -> 2x Double
val B:IndexedMemoryChunk[Double];

def scatter() {
  for (var i:Int = 0; i < nRows; ++i) {
    for (var ii:Int = 0; ii < P; ii += 16) {
      for (var jj:Int = 0; jj < nRows; jj += 16) {
        val tmin1 = min(ii + 16, P);
        for (var p:Int = ii; p < tmin1; ++p) {
          val tmin2 = min(jj + 16, nRows);
          for (var j:Int = jj; j < tmin2; ++j) {
            A(2*(i * nRows * P + j + p * nRows)) =
                B(2*(p * nRows * nRows + i * nRows + j));
            A(2*(i * nRows * P + j + p * nRows) + 1) =
                B(2*(p * nRows * nRows + i * nRows + j) + 1);
} } } } } }
```

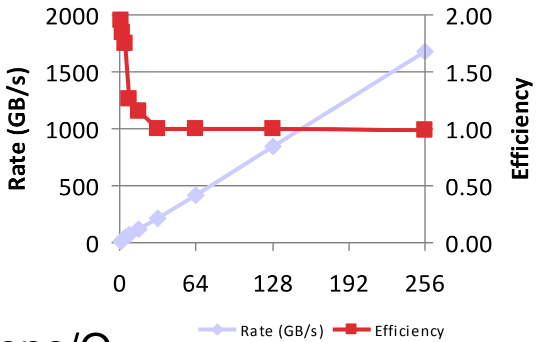
FFT after X10 2.4.0

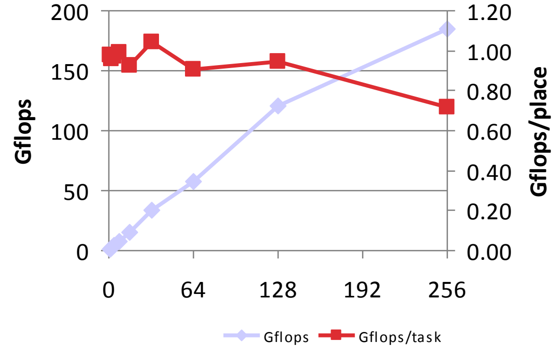
```
val A:Array_2[Complex];
val B:Array_2[Complex];

def scatter() {
  for (i in 0..(nRows-1))
    for (var ii:Long=0; ii<P; ii += 16)
      for (var jj:Long=0; jj<nRows; jj += 16)
        for (p in ii..(min(ii+16,P)-1))
          for (j in jj..(min(jj+16,nRows)-1))
            A(i, nRows*p+j) = B(nRows*p+i, j);
          }
        }
      }
    }
  }
}
```

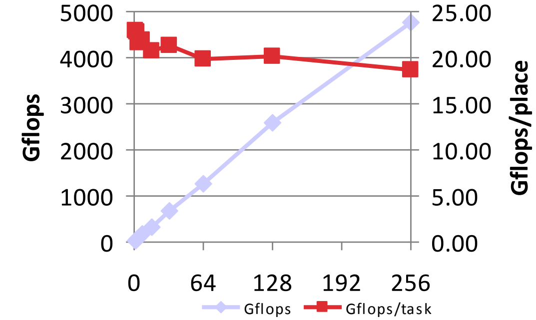
X10 Performance on Classic HPC Kernels

STREAM | FFT | HPL

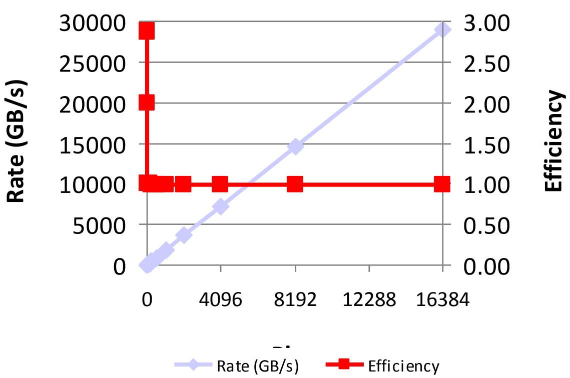
Power 775

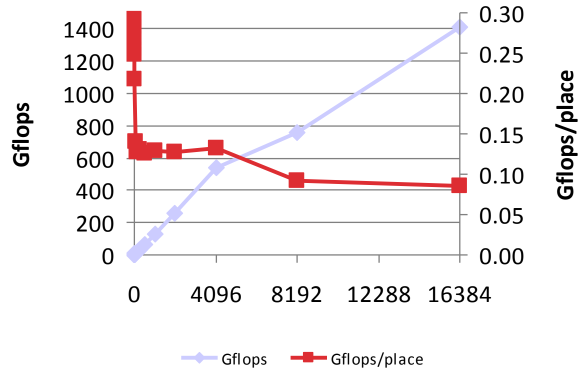


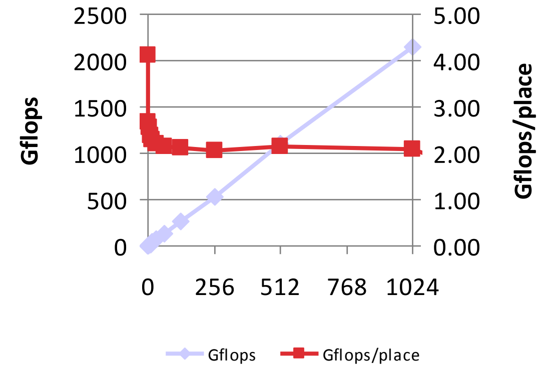




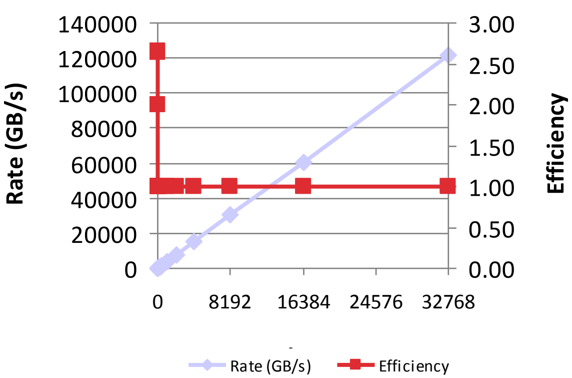
BlueGene/Q

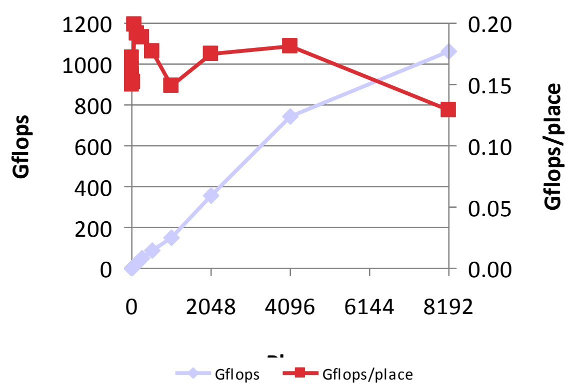


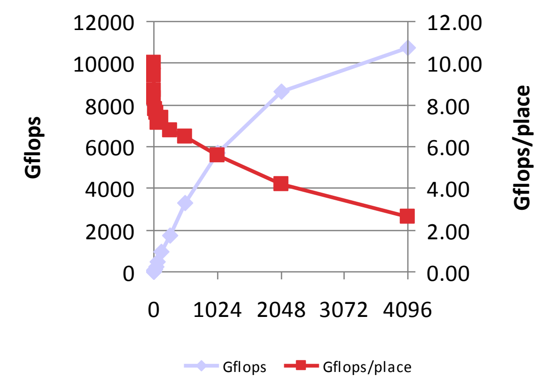




K Computer

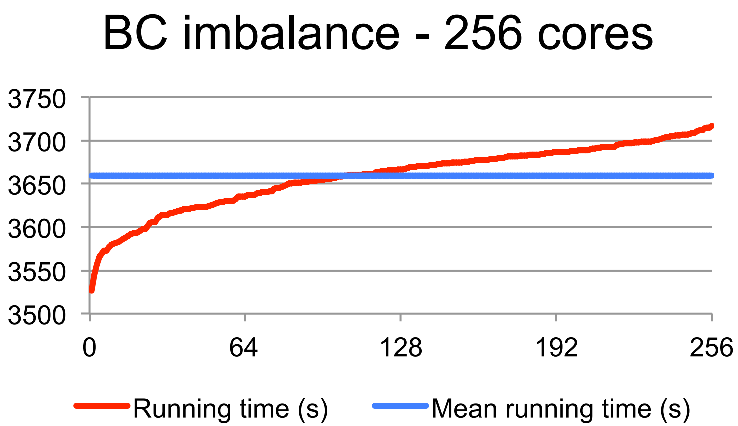






Irregular kernels

- Unbalanced Tree Search
  - count nodes in randomly generated tree
    - using a crypto algorithm
  - trillion of tasks, unknown count
  - highly irregular, intractable if not balanced
  - locality-insensitive



- Betweenness Centrality
  - measure the betwenness centrality of each node in an R-MAT graph
    - parallel single-source shortest-paths starting from each vertex in the graph + reduction
  - few tasks, known count
  - reasonably balanced if nodes are randomly permuted and equally split across tasks
  - locality-insensitive (assuming a small graph replicated across tasks)

=> Two very different and very challenging state-space exploration problems

Productivity: Libraries

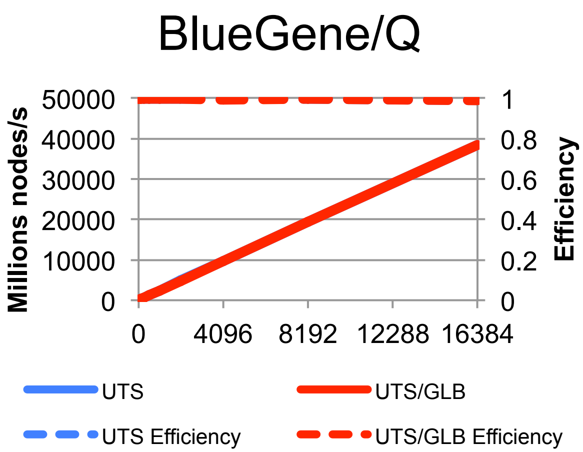
- X10's value proposition
  - implement simple, easy to use libraries
  - for sophisticated distributed control and data-structures
  - with very high performance
  - with very little abstraction overhead
  - usable from X10, C++, and Java
  - usable across a variety of platforms

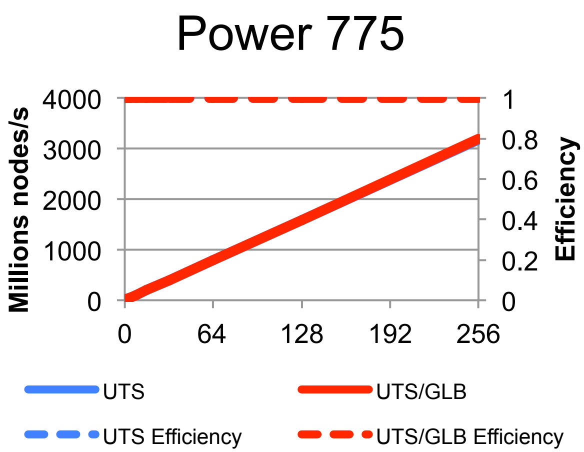
- Examples
  - Main-Memory Map Reduce
  - Global Matrix Library
  - Analytics Kernels Library
  - New in 2013: Global Load Balancing Library

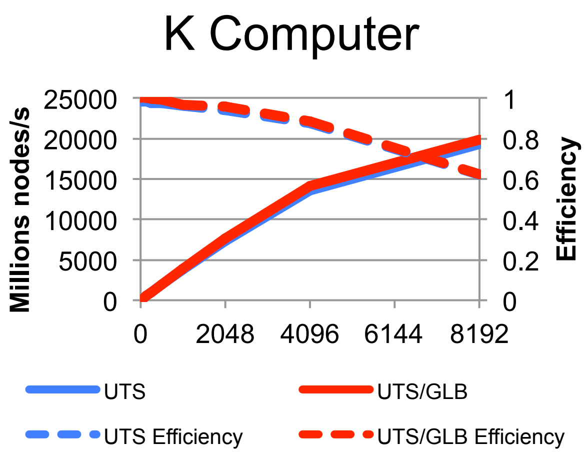
GLB Library Applied to Unbalanced Tree Search

```
// GLB defines a splittable task queue interface
public interface TaskQueue {
  public def progress(n:Long):Boolean;      // compute n steps
  public def split():TaskBag;               // divide queue into 2 halves
  public def merge(TaskBag):void;           // combine 2 queues into 1
}

// User implements the splittable task queue
public class UTSQueue implements TaskQueue { … }

// User instantiates GLB scheduler and invokes run method with root task
def runUTS(branchingFactor:Int, randomSeed:Int, treeDepth:Int) {
  val initQueue = ()=>{ return new UTSQueue(branchingFactor); };
  val glb = new GLB[UTSQueue](initQueue, GLBParameters(…));

  glb.run(rootTask(randomSeed, treeDepth)); // compute, wait for result
}
```

Unbalanced Tree Search: Performance







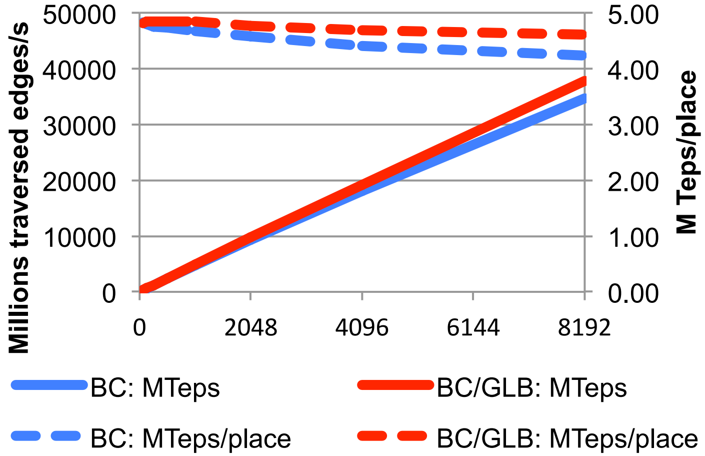
(geometric fixed law, branching factor: 4, seed: 19, depth 13 to 21)

When comparing to a sequential implementation

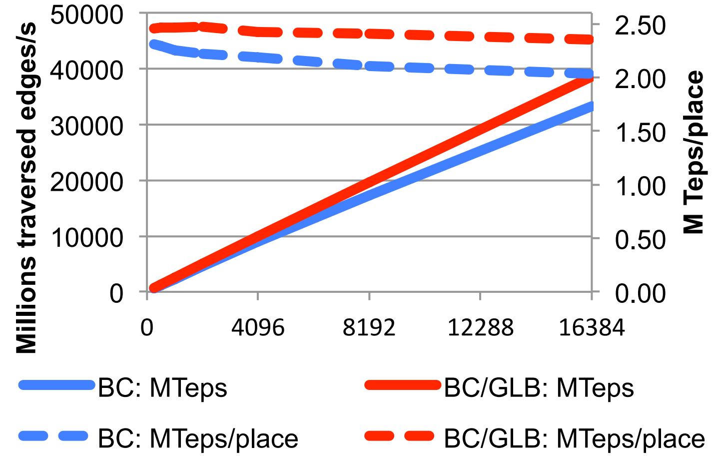
- UTS is 97.4% efficient with 16384 BG/Q cores, UTS/GLB is 97.0% efficient
- UTS is 97.9% efficient with 256 Power 7 cores, UTS/GLB is 98.5% efficient
- UTS and UTS/GLB are 96% efficient with 1024 K Computer cores
- UTS and UTS/GLB are 62% efficient with 8192 K Computer cores

Betweenness Centrality: Performance

K Computer



BlueGene/Q



(2^19 vertices, 2^22 edges)

- BC/GLB is faster than BC
  - 16384 BlueGene/Q cores: 16%, 8912 K Computer cores: 9%

- BC/GLB is more scalable than BC
  - BC with 16384 cores is 88% efficient compared to 256 cores (BlueGene/Q)
  - BC/GLB with 16384 cores is 96% efficient compared to 256 cores (BlueGene/Q)