Alternative High Performance Benchmarks

Kurt R. Rudolph   William D. Gropp   Laurance Angrave

Department of Computer Science, University of Illinois Urbana-Champaign

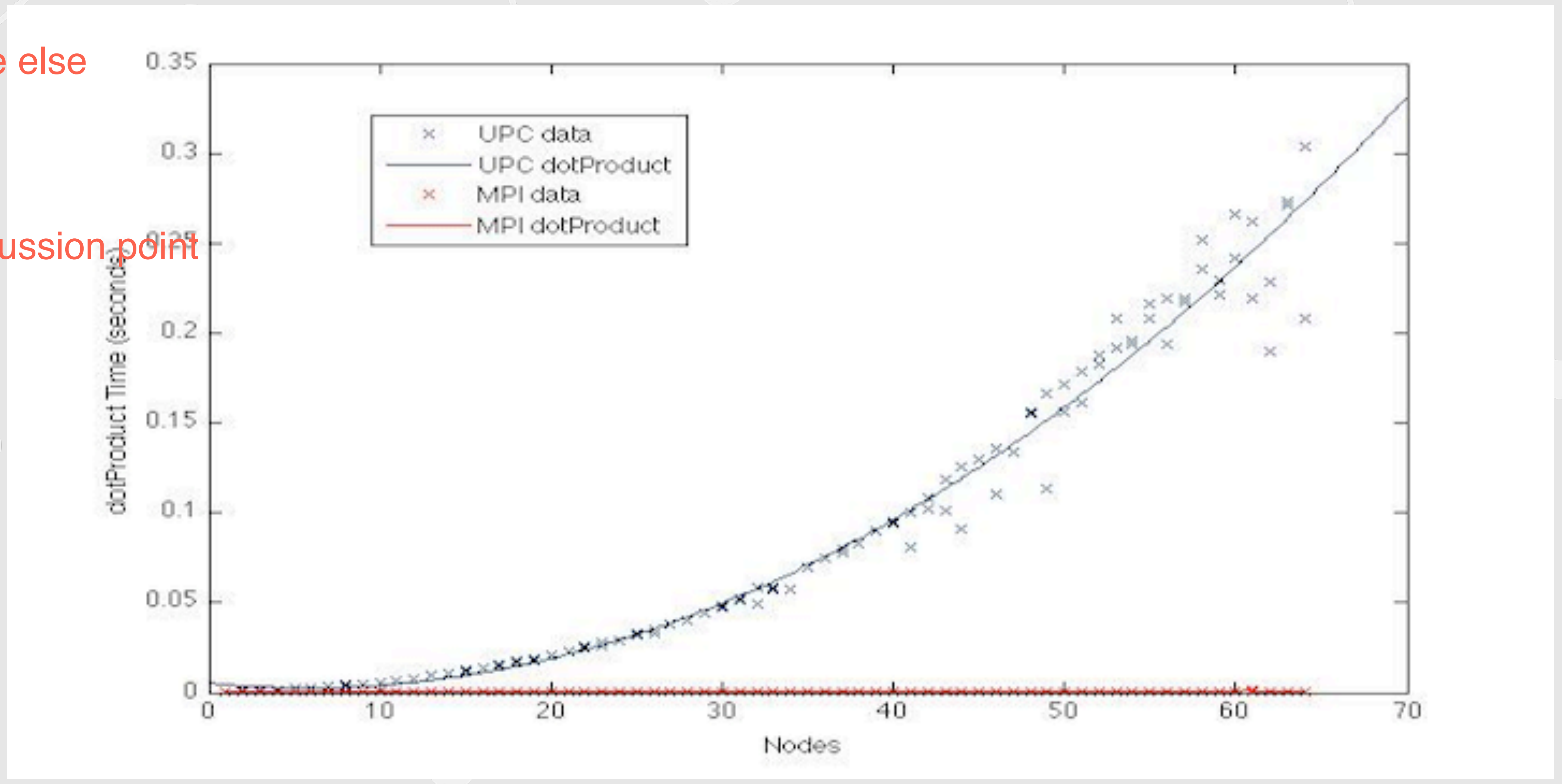
Abstract

Benchmarks for High-Performance clusters generally focus on floating-point intensive calculations. Few of these address data intensive graph operations, a class of computation increasingly growing in demand. As an alternative to standard floating-point benchmarks such as LinPack, a new benchmark, the Graph 500, has been proposed. The benchmark employ's multiple implementations with the intention to identify desecrate performance aspects via the comparative results, however a PGAS model has yet to be developed. This project focuses on early understanding of the capabilities a Graph 500 UPC implementation offers. Specifically: 1. the UPC expressibility for irregular graph operations of Graph500 2. simple performance testing the efficiency of UPC with runs up to 1024 cores.

1 The Graph 500 Benchmark

A data intensive benchmark comparing graph operations and implemented in a variety of parallel programming models, the Graph 500 provides insight into performance aspects of HPC systems not normally seen in most floating-point benchmarks.
The benchmark is composed of two kernels. The first kernel is a scalable data generator component which produces the edge tuples of the graph. The second kernel is the graph traversal component. Within the second kernel, 64 random search keys are selected from the node of the generated graph and a breadth-first search is performed for each key. Both kernels are timed and verified. get rid of
Explain General issues
The UPC implementation of this project offers the additional perspective of a partitioned global address space model and further identify discrepancies in HPC system performance.

3.1 Timing Results



2 Unified Parallel C (UPC)

A partitioned global address space (PGAS) language that extends C. Specifically, UPC employs *shared pointers* which enabling the threads of a program access to a common address space.

```
#define N 10000
shared int *vectorA;
shared int *vectorB;
shared int *vectorC;
// C = A \dot B
void dotProduct() {
 int i;
 upc_forall( i= 0;
        i< N; i++, i) {
  vectorC[i]= vectorA[i]*vectorB[i];
 }
 upc_all_reduceI(
   (shared int*) vectorC,
   (shared int*) vectorC,
   (upc_op_t) UPC_SUM,
   THREADS, 1,
   (upc_flag_t) sync_mode);
}
```

The UPC example above distributes the work of computing a dot product across all available threads in the program.

3.2 UPC Issues

- Internal consistency issues.
- Blocking index computation.
  - Great load balancing
  - Computability issues
- Mixing with MPI 2-sided communication generally leads to chaos.
  - Compiler needs to know about any current communicators to correctly calculate an index.
- Pointer arithmetic is rather costly.

go somewhere else

More of a discussion point

3.0

The results to the right were taken on the NCSA Blue-Print cluster. The cluster employs 64 nodes, each node supporting 16 power5 processing cores and 64GB of memory. Both the MPI and UPC timing results were taken utilizing 1 to 64 nodes with 16 processes per node accordingly.

[1] MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, July 18, 1997.

[2] Tarek A. El-Ghazawi, William W. Carlson, Jesse M. Draper. UPC Language Specifications V1.1 (http://upc.gwu.edu). March, 2003.

[3] William W. Carlson, Jesse M. Draper. Introduction to UPC and language specification CCS-TR-99-157.

4 The Graph500 BFS

The Graph 500 employs multiple implementations using MPI and OpenMP.
The "Simple" MPI implementation achieves two-sided communication through communicator functions such as *send* and *receive*. Sample from Graph500 BFS MPI Simple:

```
MPI_Irecv( recvbuf, coalescSize * 2, INT64_T_MPI_TYPE,
  MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &recvreq);

MPI_Isend( &outgoing[ dest * coalescSize * 2], 0,
  INT64_T_MPI_TYPE, dest, 0,
  MPI_COMM_WORLD, &outgoing_reqs[dest]);
```

The "One Sided" MPI implementation achieves one-sided communication through MPI *windows*, a feature built into MPICH2 API. When a thread creates an MPI *window* access to another threads address space is enabled through the variables associated through that window. Sample from the Graph500 BFS MPI One Sided:

```
//Open window
MPI_Win_create(pred, nlocalverts * sizeof(int64_t),
  sizeof(int64_t), MPI_INFO_NULL,
  MPI_COMM_WORLD, &pred_win);
//close window
MPI_Win_free(&pred_win);
```

The UPC implementation of this project does not require "windows" to achieve one sided communication. However, address space used in any collective operations needs to be allocated in shared memory space and receives the according performance loss of accessing a shared pointer over a local pointer. Sample from the projects' Graph500 BFS UPC implementation:

```
shared int64_t *pred2= (shared int64_t*)
  upc_all_alloc( nlocalverts * sizeof( int64_t));
shared int64)
shared int64_t *local_vertices= (shared int64_t*)
  upc_all_alloc(nlocalverts * sizeof(int64_t));

upc_all_reduceI( (shared int64_t*) pred2,
  (shared int64_t*) (local_vertices + v_local),
  (upc_op_t) UPC_MIN,
  (int64_t) (nlocalverts * THREADS),
  (int64_t) nlocalverts, NULL,
  (upc_flag_t) sync_mode);

upc_free(pred2); upc_free(local_vertices);
```