Unified Parallel C and the Computing Needs of Sandia National Laboratories

Jonathan Leighton Brown, Zhaofang Wen

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.doe.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

Unified Parallel C and the Computing Needs of Sandia National Laboratories

Jonathan Leighton Brown, Zhaofang Wen

Abstract

As Sandia looks toward petaflops computing and other advanced architectures, it is necessary to provide a programming environment that can exploit this additional computing power while supporting reasonable development time for applications. Thus, we evaluate the Partitioned Global Address Space (PGAS) programming model as implemented in Unified Parallel C (UPC) for its applicability. We report on our experiences in implementing sorting and minimum spanning tree algorithms on a test system, a Cray T3e, with UPC support. We describe several macros that could serve as language extensions and several building-block operations that could serve as a foundation for a PGAS programming library. We analyze the limitations of the UPC implementation available on the test system, and suggest improvements necessary before UPC can be used in a production environment.

Contents

1	Introduction	7
2	Sample Problem: Sorting.....	8
2.1	Algorithms Considered	8
2.2	Testing Protocol	9
2.3	Results	9
2.4	Discussion: Writing The Code	10
3	Sample Problem: Minimum Spanning Tree	12
3.1	Algorithms Considered	12
3.2	Testing Protocol	13
3.3	Results	13
4	Common Functions.....	14
4.1	Bucketing	14
4.2	Gather Buckets	15
4.3	Thread Prefix	15
4.4	Generalized Thread Prefix	16
4.5	Thread Concatenate	16
5	Macros	16
5.1	Thread View	17
5.2	Reverse Thread View	17
5.3	Block Size Mapping	18
6	Comparison to MPI	18
6.1	Communication Latency and Cost	19
6.2	Radix Sort	22
6.3	Prim's Algorithm	23
7	Future Research.....	23
7.1	Performance Model.....	23
7.2	Library Function	25
7.3	Implementation.....	25
7.4	Petaflop Architecture	27
8	Conclusions	27
9	Acknowledgements	28
	References	29

Appendix

A	Comparative Sorting Algorithm Results	31
B	Summary of Sorting Results By Algorithm	37
C	Minimum Spanning Tree Results	41
D	Communication Latency and Cost Measurements	47
E	Code Samples	53
E.1	Sorting Algorithms	53
E.2	Minimum Spanning Tree Code	76
E.3	Low Level Code Tests	95

Figures

A.1	Results for Sorting a 128K-Entry List	31
A.2	Results for Sorting a 512K-Entry List	32
A.3	Results for Sorting a 2M-Entry List	33
A.4	Results for Sorting a 4M-Entry List	34
A.5	Sorting Times in Processor Cycles for 8M-Entry List	35
B.6	Sorting Results for Parallel Bubble Sort	37
B.7	Sorting Results for Bottom-Up Merge Sort	38
B.8	Sorting Results for Radix Sort	39
B.9	Sorting Results for Sample Sort	40
C.10	Time in processor cycles for each implementation of Prim's algorithm - 10,000- node graph	41
C.11	Speedup for each implementation of Prim's algorithm - 10,000-node graph . . .	42
C.12	Time in processor cycles for each implementation of Prim's algorithm - 40,000- node graph	43
C.13	Speedup for each implementation of Prim's algorithm - 40,000-node graph . . .	44
C.14	Time in processor cycles for each implementation of Prim's algorithm - 160,000- node graph	45
C.15	Speedup for each implementation of Prim's algorithm - 160,000-node graph . .	46
D.16	Average Cost per Element for Writes to a UPC Array	47
D.17	Access Time Per Array Element for Shared Array Reads	48
D.18	Average Cost per Element for Reads from a UPC Array	49
D.19	MPI Communication Cost: Direct Transfer	50
D.20	MPI Communication Cost: Collective Operation	51
D.21	T3E Cache Profile	51

Tables

1	Run Times for Sequential Radix Sort in Cycles	10
2	Cost per Element for Reads from a UPC Array	20
3	Time for Consecutive Unoptimized Reads from a UPC Shared Array	21
4	Execution Time of MPI Radix Sort in Processor Cycles	22
5	Scaling of MPI Radix Sort	23
6	Run Times for 40,000-Node Graph	23

Unified Parallel C and the Computing Needs of Sandia National Laboratories

1 Introduction

Unified Parallel C (UPC) is an extension to standard C to include explicit parallel directives in the language [3]. UPC programs are designed to be executed concurrently across multiple processors or processing elements, referred throughout as threads. UPC is an implementation of the Global Address Space (GAS) model; each thread has a portion of its memory space that is private to it and a portion that is globally addressable. Each thread can access any address in globally addressable memory on any other thread. However, the language has an affinity attribute for this globally-addressable memory so that a particular thread “owns” a segment of the globally addressable memory [2]. Although UPC provides a distributed shared memory interface to the programmer, there exist compilers and other tools for converting UPC code to MPI and other underlying network layers. Currently compilers are available from Cray, SGI, HP/Compaq, and UC-Berkeley.¹

The UPC language extensions are designed to be unobtrusive and intuitive. The **shared** keyword is used to declare a globally-addressable variable. As UPC is array-oriented, there is a parameter to the shared declaration that describes the block size. If an array has block size b , entries $0 \dots b-1$ have affinity to thread 0, entries $b, \dots, 2b-1$ have affinity to thread 1, and so on, so that the array is striped across globally-addressable memory [3]. This differs from Co-Array Fortran, where arrays with co-array dimensions are replicated across all threads [7]. Macros described in section 5 can be used to convert the UPC view of an array into that of Co-Array Fortran. We shall throughout abuse the language and call the globally-addressable memory declared with the **shared** keyword “shared”, with the understanding that this is a logical view as shared, irrespective of the underlying hardware.

Other language extensions include those expected of a parallel language like OpenMP, with some new ideas. UPC has a **upc_forall** directive for compiler-parallelized loops, with an additional argument so that the programmer can suggest which thread should be assigned which loop iteration based on affinity of a shared variable. In addition to a barrier for synchronization, UPC has a split-phase barrier implemented with **notify** and **wait** keywords, so that work can be done between notifying other threads of reaching the barrier and waiting for all threads to reach the barrier.

Recent work in UPC standards groups has focused on adding MPI-style collective operations to UPC [9]. These would include gather and scatter operations adapted to the UPC memory model. Although our research suggests that these could be beneficial from a

¹For more details on availability of UPC compilers, or for copies of UPC standards and specifications, visit the official UPC Consortium website at upc.gwu.edu.

performance perspective (see Section 6), they were unavailable on test systems and thus not evaluated.

What follows is a summary of research done to date on UPC. Sample problems are used to test language features and evaluate performance of UPC on test systems. Common functions abstracted from test programs for the sample problems are described. Macros defined to augment or vary the presentation of UPC are described and defined. We then make a comparison of UPC to MPI, and conclude with a discussion of current and future lines of research before providing some concluding remarks. An appendix of code developed is included at the end of this document.

2 Sample Problem: Sorting

The first sample problem considered was that of sorting a list of numbers. For our purposes, we restrict ourselves to positive integers generated uniformly at random. We implement a variety of algorithms in UPC with different communication patterns and develop two good sorting algorithms for UPC.

2.1 Algorithms Considered

Six sorting algorithms were implemented in UPC, two in each of three broad categories. QuickSort and a merge sort variant using Batcher’s odd-even operators were implemented with fine-grained parallelism, using extensive synchronization, collective recursive function calls, and/or UPC parallel loops. Radix and random sample sorts were implemented using coarse parallelism similar to message-passing or distributed-memory code. A bottom-up merge sort and a “parallel bubble sort” were implemented as well, similar to the radix and sample sorts but not as tuned for performance. We now briefly detail each implementation. Throughout N denotes the length of the list of numbers to be sorted and P denotes the number of threads.

- QuickSort: A recursive formulation of QuickSort was used, wherein all threads executed recursive calls concurrently. The partitioning of the list of numbers was done in parallel, with each processor partitioning an equal-sized subset of the list. Barrier synchronization was used to maintain the collective nature of the function calls.
- Merge Sort with Batcher’s Odd-Even: This was an iterative merge sort, implemented with UPC `upc_forall` loops for compiler parallelization. As the straightforward implementation with Batcher’s odd-even operators had several nested loops, and `upc_forall` loops cannot be nested, implementations with the UPC parallelization at each level of nesting were tried. No appreciable performance benefits were measured when the level of nesting was varied. Code was adapted from [8].

- Bottom-up Merge Sort: Each processor locally sorted an $\frac{N}{P}$ fraction of the list using an efficient serial radix sort. These sorted lists were then merged in a tree-like fashion.
- Parallel Bubble Sort: This was like the bottom-up merge sort, except that, instead of merging the list in a tree-like fashion, each processor merged and split its portion of the list with its neighbors in $P - 1$ rounds.
- Radix Sort: This sort has parameters r , the radix, and d , the number of rounds. If M is the maximal number in the list, then $d = \lceil \log_r M \rceil$, so only one need be specified. The sort proceeds from least to most significant digit, so that on the i^{th} iteration, each thread chooses a contiguous block of $\frac{N}{P}$ numbers, sorts these into buckets based on the i^{th} least-significant digit. The buckets are then collected into a shared array in order – so that all 0 buckets from threads 0 to $P - 1$ are concatenated and placed before all 1 buckets from threads 0 to $P - 1$, and so on. This is repeated for d iterations, at which time the list is sorted.
- Random Sample Sort: There is an oversampling factor, s , that is used to smooth the random sample. Each thread chooses a contiguous block of $\frac{N}{P}$ numbers from the list and selects s samples from this block. The sP samples are collected, sorted, and, from these, $P - 1$ pivots $p_1 < p_2 < \dots < p_{P-1}$ are selected. Each thread then buckets its numbers into each of P buckets, based on which pivots they fall between. These buckets are then concatenated as with the radix sort. Thread i then takes the numbers between p_i and p_{i+1} and locally sorts these with an efficient serial radix sort. Code was adapted from [1].

Code for all sorting algorithms is included in Section E.1.

2.2 Testing Protocol

All tests were done on a Cray T3e/750 with 1, 2, 4, 8, 16, or 32 processing elements. The “random” list was generated in advance with the `rand()` system call and saved to a file. All sorting algorithms were implemented as function calls, and times were computed using the `times()` system call on thread 0 before and after the call to the sorting function. Times are reported in absolute clock cycles. Due to restrictions on the size of shared memory segments of UPC programs on the Cray, only lists of sizes between 128K and 16M numbers were used.

2.3 Results

The sorting algorithms were quickly differentiated by their scaling and execution time (Figures A.1 - A.5). The fine-grained QuickSort and merge with Batcher’s odd-even operators were an order of magnitude slower than the other implementations. The QuickSort implementation in UPC required two processors, so scaling from a UPC uniprocessor time is not

available, but the Odd-Even implementation showed almost no scaling as the number of processors increased. These two implementations reached the shared memory segment limit on the test system quickly, and so measurements are only available for lists of up to 512K entries.

The bottom-up merge and parallel bubble sorts showed anemic scaling. Although they performed reasonably well for small numbers of processors, their scaling peaked between 4 and 8 processors. This is inherent in the algorithms – both contain a series of exchange steps that grows with the number of processors, $\Theta(P)$ for the parallel bubble sort, and so impacts scaling.

The best execution times were demonstrated by the radix and random sample sorts for all list lengths. Both showed good scaling with increasing numbers of processors, with the random sample sort doing somewhat better. When considering scaling problem size, the radix sort showed the most promise as it showed sub-linear scaling with increasing problem size for 16 and 32 processors, whereas the other sorts all showed at least a linear increase in processing time when increasing the list length (Figure B.8).

We also compare our results to a sequential radix sort. Sequential times are provided for lists of length ranging from 128K to 4M entries in Table 1.

Table 1. Run Times for Sequential Radix Sort in Cycles

<i>List Length</i>	<i>Running Time</i>
128K	3.85E+07
512K	1.54E+08
2M	6.16E+08
4M	1.25E+09

At 128K, only 32 processors for radix and random sample sorts were faster than the sequential implementation. At 512K or 2M, at least 16 processors for radix or 32 for random sample are needed to be faster than the sequential implementation. For 4M, at least 16 processors are needed for either radix or random sample.

2.4 Discussion: Writing The Code

We examine the radix sort, as it was the most analyzed and had the most aberrant behavior on initial runs. The code is included in Section E.1.6.

The radix sort was written to use GAS techniques wherever possible. The first idea was to maintain perfect load balancing for each iteration of the radix loop. Thus, instead of identifying threads with buckets and sending all the “0” buckets to thread 0, all the “1” buckets to thread 1, and so on, a large shared array equal in length to the list was employed

as a buffer. Threads compute where each of their buckets are to go in the buffer, copy them to the buffer, then select a new contiguous block from the buffer after synchronization. Buckets are copied back so that, in the i^{th} round, bucket for digit j on thread t will be copied back so that buckets appear in sorted order first by digit and then by thread. Threads take contiguous blocks in order, so that thread 0 takes the first $\frac{N}{P}$ items, thread 1 takes the second $\frac{N}{P}$ items, and so on. This preserves the radix ordering for a least-significant-bit radix sort. It became clear that the original list was unused except at the start when copied from and at the end when copied back to, and so the original list was used as the buffer. This helped with the memory restrictions on the shared segment.

Synchronization was then found to be a bottleneck. The original code had many barriers, used mostly for parallel prefix operations. Although the code still has a single thread computing the number of iterations (“depth”) of the radix sort before the start of the main loop, inside the main loop, computation of `TargetOffsets` is now done independently by each thread. This allowed for the removal of an additional barrier statement from the main loop, leading to a measurable performance gain. Changing the depth code did not measurably improve performance. In assembly code, the barrier statement is converted to over fifty instructions, including multiple calls to external UPC runtime functions. Thus, a style of coding preferring re-computation to synchronization was favored.

The last significant change to the radix code was the introduction of the `OVERDEC` macro. In our original implementation, the radix was hardwired to P , except if $P = 1$, in which case it was set to 2. This meant that the radix for 32 processors was only 32. A serial radix implementation on the Cray showed empirically that a radix of between 128 and 512 was best for performance. The serial code showed large performance gains on doubling the radix, up to around 256, at which point it leveled off before falling due to excess overhead. This in part explains the super-linear speed-up observed in our initial study: doubling the number of processors also doubled the radix, slashing the number of iterations of the main loop. Thus, the `OVERDEC` macro is used to make the radix very large at the outset. Although it was not explored in detail, the code can also be written so that the number of iterations of the main loop are parameterized instead of the radix, and thus one could demand two iterations of the loop with the smallest possible radix. This would be, in essence, a smart version of the sample sort. This was not pursued.

The major innovation of the UPC implementation is the load balancing. Using the shared array as a buffer, we achieve perfect load balancing throughout the execution of the program, in that each thread always holds an $\frac{N}{P}$ fraction of the list. This final code could be improved with collective operations, similar to those in MPI, to streamline the copy back in the main loop.

3 Sample Problem: Minimum Spanning Tree

A second sample problem considered was that of finding a minimum spanning tree of an undirected graph with weighted edges. For the graph representation, we use adjacency lists with a structure for each entry with the vertex number and edge weight.

3.1 Algorithms Considered

A minimum spanning tree algorithm takes as input a graph $G = (V, E)$, with a weight function wt for each edge, and outputs a tree T and weight w . We take $V = \{1, 2, \dots, |V|\}$. The three most common MST algorithms in the literature are:

- Prim's [5]:
 - Set $T = \{0\}$. Set $new = 0$. Set $w = 0$. For each $v \in V \setminus T$, set $dist(v) = \infty$.
 - For $i = 1, \dots, |V| - 1$:
 - * For each vertex $v \in V \setminus T$, if $\{v, new\} \in E$, set $dist(v) = \min\{dist(v), wt(\{v, new\})\}$.
 - * Choose $v' \in V \setminus T$ such that $dist(v') = \min\{dist(v) | v \in V \setminus T\}$. Set $new = v'$.
 - * $w = w + dist(new)$. Add new to T .
- Boruvka's [4]:
 - Make a list L of $|V|$ trees, each initially a single vertex.
 - While L contains more than one tree:
 - * For each tree S in L , select a minimum weight edge e that connects S to $G \setminus S$.
 - * Add each e to our minimum spanning tree, T , merging trees in L as they become connected
- Kruskal's [4]:
 - Let $T = (V, \emptyset)$. Let $\mathcal{E}$ be the set of edges E as a list.
 - Sort $\mathcal{E}$ by increasing order of weight.
 - For each $e \in \mathcal{E}$ (in order), if $e = \{x, y\}$ and x and y are separated in T , add e to T .

Parallel versions of Prim's and Boruvka's algorithms were implemented in UPC. As suggested by [5], we parallelized Prim's algorithm by dividing the vertices of G among processors $p_1, p_2, \dots, p_P$ as $V_1, V_2, \dots, V_P$, computing the minimum-distance v'_j from each V_j in parallel, and then from these v'_j , computing a minimum-distance v' . There are four variants of the UPC implementation:

- Prim-UPC-1: A naive implementation of the algorithm where each V_j is chosen to be of size $\frac{N}{P}$, where $N = |V|$ and P is the number of processors.
- Prim-UPC-2: A change to Prim-UPC-1 such that each V_j is chosen to evenly distribute the edges of G , instead of the vertices, as the main loop of Prim’s algorithm checks the edges at each unadded vertex.
- Prim-UPC-3: A change to Prim-UPC-2 so that all V_j are updated periodically using a fixed schedule. This load balancing is done irrespective of the actual distribution of work in the system, and thus will be called “oblivious load balancing”.
- Prim-UPC-4: A change to Prim-UPC-3 so that the V_j are updated only when there is a processor that has exhausted its V_j (i.e., all are in T).

In our parallel version of Boruvka’s algorithm, every vertex selects a lightest-weight edge for contraction. These are selected with a secondary key of vertex number to prevent cycles. The connected components are then identified, and the induced graph resulting from contracting all components to a single vertex is then solved recursively.

We make the simplification that our implementations only return w , not w and T . Minor modifications would make them return T as well. We assume all inputs are adjacency lists.

An MPI version of Prim’s algorithm was implemented for comparison. See Section 6.3 for details.

3.2 Testing Protocol

All tests were done on a Cray T3e/750 with 1, 2, 4, 8, or 16 processing elements. Graphs with 10,000, 40,000, and 160,000 nodes were constructed by connecting vertex i to those vertices j where j is prime and j divides i . A randomized parameter p was also introduced for some test graphs so that edges were only added with probability p . For the results presented in this paper, we take $p = 1$. Times are reported in absolute clock cycles.

3.3 Results

The oblivious-rebalancing UPC implementation, Prim-UPC-3, showed near-linear speedup, and except for single-processor tests, had the best time for any of the UPC implementations. The Prim-UPC-2, with initial load balancing, and Prim-UPC-4, with monitored load balancing, achieved some speedup, of between 8 and 11 for 16 processors. The naive UPC implementation showed very poor performance, slowing down for two processors and eventually achieving a speedup of roughly 5 for 16 processors. For all implementations and all number of processors, the ratio of time for the 40,000-node graph to that for the 10,000-node

graph, or for the 160,000-node graph to that for the 40,000-node graph was 16 on average, and varied little.

For the 10,000-node graph, times are presented in Figure C.10 and scaling results are presented in Figure C.11. For the 40,000-node graph, these are Figures C.12 and C.13. For the 160,000-node graph, these are Figures C.14 and C.15.

Performance for the UPC implementation of Boruvka’s algorithm was poor. For a 10,000-node graph, it took 4.48E+11 processor cycles on a single processor, 2.38E+11 for four processors, and 1.58E+11 for sixteen processors. These are two orders of magnitude greater than for any UPC implementation of Prim’s algorithm on the same graph for the same number of processors. Trials on larger graphs did not terminate after five days. The scaling was very shallow – for this 10,000-node graph, only 2.83 for 16 processors.

4 Common Functions

Several operations were extracted from the sorting algorithm implementations and implemented as common functions. These are the first steps toward a UPC library or programming idiom.

4.1 Bucketing

Several of the sorting algorithms use an idea similar to bucketing – assigning each element of an array to one of several logical buckets.

- Declaration:

```
void Bucketing(int * localSrc, int * localDest, int * localBuckets[], int length, int
* Count, int range, int (*getkey)(int))
```

- Input:

- *localSrc* - array of elements to be assigned to buckets
- *length* - length of *localSrc* array
- *range* - maximum value returnable by *getkey*
- *getkey* - function that assigns each element of the source array to a bucket

- Output:

- *localDest* - array of elements assigned to buckets, serialized
- *localBuckets* - array of pointers into *localDest* such that *localBuckets*[*i*][*j*] returns the j^{th} item in the i^{th} bucket
- *Count* - count of items in each bucket

4.2 Gather Buckets

Once items have been placed into local buckets, we would like to gather them to a shared array so that they can be accessed by all threads. Buckets are arranged in the array first by bucket number and then by source thread.

- Declaration:

```
void GatherBuckets(int * localBuckets[], int length, int * Count, int range, shared
int * list)
```

- Input:

- *localBuckets* - array of pointers from **Bucketing** call
- *length* - number of items contained in all buckets
- *Count* - array of bucket lengths as returned by **Bucketing**
- *range* - maximal bucket number

- Output:

- *list* - shared list with items copied from *localBuckets* on each thread and placed in order first by bucket number and then by source thread

4.3 Thread Prefix

For our parallel prefix operation, each thread t holds a value S_t , and we compute $T[t]$ to be the sum of S_0 to S_t . This function is called by all threads, and, as it was implemented, was a synchronization point.

- Declaration:

```
shared int * ThreadPrefix(int S) or void ThreadPrefix(shared int * target_indices,
int S)
```

- Input:

S - value held by the thread

- Output:

`target_indices` or return value - array of values such that the i^{th} value is the sum of those S values held by threads 0 to i .

4.4 Generalized Thread Prefix

A generalization of the thread prefix operation was developed that added a function pointer to the input. This allowed an arbitrary binary function from integers to integers to be used for the parallel prefix operation, not just addition. This was frequently used with a maximum or minimum operation.

4.5 Thread Concatenate

This allows us to serialize several private arrays held by different threads into a large, shared array, thus making all information available to all threads. This is similar to the gather operation in MPI.

- Declaration:

```
void ThreadConcatenate(shared int * target, int * Source, int S)
```

- Input:

Source - a list of values held by the thread

S - the length of *Source*

- Output:

target - shared array into which all private arrays are to be copied

5 Macros

UPC pointers have block, offset, and phase fields that are determined for each memory location in the array at declaration. As UPC shared arrays are striped across the different threads, it is often advantageous to access a UPC shared array as if it had a different block size. The following macros are used to manipulate UPC shared array indices to produce a malleable interface to the shared array, presenting different blocking patterns to the application programmer.

5.1 Thread View

This macro produces an index j into an array with block size b given that we have P threads and we want the i^{th} item (physically) mapped to thread t . This allows UPC to simulate the Co-Array Fortran view of globally-addressable memory [7].

- Notation:

$$j = ThreadView(t, i, b, P)$$

- Input:

t - thread identifier

i - offset on thread

b - block size of array

P - number of threads

- Output:

j - array index

- Definition:

$$j = \lfloor i \div b \rfloor \star P \star b + t \star b + (i \bmod b)$$

5.2 Reverse Thread View

This macro produces a thread number t and offset i given an index j into an array with block size b and P threads. t and i are defined such that, on input of these to the Thread View macro with the same parameters b and P , the Thread View macro returns j .

- Notation:

$$(t, i) = ReverseThreadView(j, b, P)$$

- Input:

j - array index

b - block size of array

P - number of threads

- Output:

t - thread identifier

i - offset on thread

- Definition:

$$t = \lfloor j \div b \rfloor \bmod P$$

$$i = (j \bmod b) + \lfloor \lfloor j \div b \rfloor \div P \rfloor \star b$$

For implementation, it is necessary to create separate “thread” and “offset” macros to return t and i separately.

5.3 Block Size Mapping

This macro produces an index j into an array with block size β given an index i into an array with block size b , given that we have P threads. This is particularly useful for providing the application programmer with arbitrarily large UPC array blocks, whereas the implementation of the UPC pointer as a 32- or 64-bit value limits the block size possible at declaration. For example, on the Cray T3e, 10 bits are reserved for the block size, and so it is only through a software solution that a sixteen-million-entry array can be divided into two-million-entry contiguous chunks of the array for each of eight threads.

- Notation:

$$j = \textit{BlockSizeMapping}(i, b, \beta, P)$$

- Input:

i - array index for source block size

b - source block size

β - target block size

P - number of threads

- Output:

j - array index for target block size

- Definition:

$$j = \textit{ThreadView}(\textit{ReverseThreadView}(i, b, P), \beta, P)$$

6 Comparison to MPI

UPC is a language implementation of the Global Address Space programming model. Thus, we compare it to the current programming model, message passing, as implemented in MPI.

6.1 Communication Latency and Cost

Simple tests were implemented for UPC and MPI to measure simple communication latency and cost. We first tested the cost to write to a UPC shared element. We measured this by allocating a large (up to 2^{22} entries) shared array and measuring the time for thread 0 to traverse the array, writing a constant to each array location. To simulate network congestion, we also timed thread 0 when all threads write to the array and then synchronize at a barrier. The write test was compiled with optimization (`-O3`). The number of threads varied from 1 to 32. Overhead was computed and removed by timing the same code without the write to shared instruction. Throughout, we used a block size of 1; experiments with different block sizes showed similar results as all accesses are to individual array entries.

For writing to the shared array, no appreciable difference was observed in these measurements as the number of threads varied between 1 and 32 without network congestion. The average time in cycles to write to an element of a shared array was between 85 and 90 cycles. With network congestion, there is a slight trend toward a higher cost to write to shared with more processors, 119.92 cycles per element for a single thread and between 125 and 127 cycles per element for more than one thread, suggesting only a small penalty for a remote UPC write with network congestion. As the code with all threads writing included a barrier, the difference in single thread cost per write is most likely due to some barrier cost included in the timing measurement. These results are summarized in Figure D.16.

A read test was similarly devised. We again traversed a large shared array, reading the shared array value and writing it to a local variable. The number of threads varied from 1 to 32. The code was similarly optimized (`-O3`), and loop and other overhead was discounted by timing the same code but writing the index variable value to the local variable instead of the shared array value.

Locality of access did matter for read accesses to shared variables. For a single UPC thread, reading from the shared array averaged 197.66 cycles per element. For additional processors, the average time to access an array element is modeled by

$$\text{Average Cost} = \frac{1}{P}\beta + \frac{P-1}{P}\alpha \quad (1)$$

where P is the number of threads and we treat β and α as constants representing the cost to access local or remote shared items, respectively. We take advantage of the fact that the array is evenly distributed across the threads in this formula. Without congestion, $\beta = 197.66$ cycles per element. α can be approximated as a constant, but grew slightly from 496 cycles per element to 554 as the number of processors increased. The data suggest that there is a fixed cost for accessing remote shared memory, and a small term that grows as $\Theta(\sqrt{P})$, so that $\alpha = \alpha' + \Theta(\sqrt{P})$, where α' is in fact a constant.

This model works well for describing timing behavior observed with the all-to-all shared array read used to simulate network congestion. In this case, $\beta = 206.69$ cycles per element,

Table 2. Cost per Element for Reads from a UPC Array

Array Size	Thread 0 Reads				All Threads Read			
	P = 1	P = 4	P = 16	P = 32	P = 1	P = 4	P = 16	P = 32
$2^{10} = 1024$	198.72	421.41	512.50	544.48	207.71	419.75	466.06	566.83
2048	196.32	423.93	510.49	542.75	205.90	418.16	520.70	580.35
4096	196.30	424.45	511.51	544.71	205.47	418.69	515.24	586.90
8192	199.13	425.77	511.07	543.32	206.12	423.94	511.88	589.73
16384	197.83	423.42	510.50	543.45	207.81	429.03	510.57	594.63
32768	197.24	422.91	510.68	541.89	206.90	433.55	508.96	592.87
65536	197.72	422.92	510.31	542.58	206.56	432.12	511.53	594.10
131072	197.69	422.86	510.23	542.43	206.70	432.26	514.77	595.69
262144	197.75	423.17	510.28	542.54	206.73	432.34	519.96	595.81
524288	197.74	423.05	510.21	542.48	206.70	432.46	520.62	595.99
1048576	197.76	423.05	510.24	542.56	206.76	432.53	521.01	596.63
2097152	197.76	423.10	510.29	542.49	206.76	432.43	520.72	597.05
$2^{22} = 4194304$	197.76	423.10	510.29	542.46	206.76	432.57	520.74	597.30
Trimmed Mean	197.66	423.27	510.54	542.87	206.69	428.92	515.97	592.70
α Estimate		498.47	531.39	554.00		502.99	536.59	605.16

and α increased from 503 to 605 cycles per element. These results are summarized in Figure D.18.

An unoptimized version of the read test was conducted on a single thread, so that all accesses would be to local shared memory. To test caching of UPC shared items, the read from shared memory was repeated to provide two consecutive reads from the same shared memory location. When the loop overhead is removed, we find that the cost of two consecutive reads to the same location is almost exactly twice the original time. Results are shown in Table 3 below. We observe a slight trend toward a cheaper cycles per array element for the read, but due to limitations on the UPC shared segment size, we could not see whether the trend continued or stayed near 250 cycles per array element. From this we conclude that there is no caching of any UPC shared items.

By way of comparison, we implemented a simple MPI latency test. Blocking send and receive pairs were used to transmit an integer from process 0 to process 1, which replied with an integer. The time was measured from before process 0 transmitted to after it received the reply, and thus represents a round-trip time. The code was compiled with optimization (-O3). The average round-trip time was 4728 processor cycles.

Additional MPI tests were conducted to determine how the ability to move larger chunks of data per communication in MPI provides a performance advantage. Processor 0 transmitted a large array of integers – up to 2^{23} entries – to each of 31 other processors as a single message, and each processor responded with an acknowledgement message. The round-trip time was measured in cycles by processor 0, and these were averaged across the 31 destina-

Table 3. Time for Consecutive Unoptimized Reads from a
UPC Shared Array

List Length	Average Time to Read Array	Average Time for Two Reads	Ratio
$2^{10} = 1024$	2.66E+05	5.19E+05	1.95
4096	1.06E+06	2.08E+06	1.95
16384	4.17E+06	8.26E+06	1.98
65536	1.66E+07	3.30E+07	1.98
262144	6.53E+07	1.31E+08	2.01
1048576	2.63E+08	5.26E+08	2.00
$2^{22} = 4194304$	1.05E+09	2.10E+09	2.00

tions. The data are summarized in Figure D.19. Up to a payload of roughly 2^{19} integers, the data closely fit the equation

$$\text{Round-Trip Cost (Cycles)} = 5.08 \cdot \text{Number of Integers Transferred} + 3751.2$$

Thus, the cost per additional item transferred is on the order of 5 cycles once the initial communication overhead is paid. However, for larger payloads, the data closely fit the equation

$$\text{Round-Trip Cost (Cycles)} = 1.89 \cdot \text{Number of Integers Transferred} + 203711$$

Thus, we see that this long protocol has a much higher overhead, but yields a cost per additional item transferred on the order of only 2 cycles.

We also measured the time for the MPI `MPI_Allgather` collective communication, wherein portions of a large array are exchanged. Array sizes varied up to 2^{22} entries. The data are summarized in Figure D.20. The cost in processor cycles for the collective operation was modeled by:

$$\text{Round-Trip Cost (Cycles)} = 8.85 \cdot \text{Total Integers Transferred} + 189562$$

Thus, we see a slightly higher cost per additional item transferred – on the order of 9 cycles – but still far less than the cost for transferring an individual item in UPC.

For reference, the average cost per array access when traversing a large array in an unoptimized sequential C program on the test system was 6.1 cycles – 5.7 for writing a constant to each entry, 6.5 for reading the array value and assigning it to a temporary variable after loop and other overhead is discounted. With optimization, this reduces to 2.5 cycles. However, this amortized cost is affected by cache effects. Using the cache profiler

from [6], we found the cost of accessing a non-cached block of private memory is 57 cycles. These results are summarized in Figure D.21.

From this we draw the following conclusions: Respecting locality of reference on the test system provides a considerable performance boost (a factor of 3), but the UPC overhead for a local UPC access lags the cost per access for a sequential-C cache miss by almost the same factor. Although UPC is faster than MPI by an order of magnitude for transferring an individual item, MPI is able to amortize communication overhead by transferring larger packets and with communication collectives. Individual memory accesses were required of UPC programs on the test system, and show an area needing improvement.

Code for all UPC and MPI tests is included in Section E.3.

6.2 Radix Sort

An MPI version of the UPC radix sort was implemented. MPI collective operations are used for all communication steps. The MPI implementation, included in section E.1.8, is almost a direct translation of the UPC code. The only difficulty is that the MPI version cannot use the UPC shared array for automatic load balancing, so considerably more work is done inside the main loop to achieve load balancing. Run time and scaling numbers are provided in Tables 4 and 5.

Table 4. Execution Time of MPI Radix Sort in Processor Cycles

Processors	List Length						
	131072	262144	524288	2097152	4194304	8388608	16777216
1	8.14E+07	1.63E+08	3.25E+08	6.50E+08	1.30E+09		
2	4.14E+07	8.22E+07	1.64E+08	3.28E+08	6.55E+08	1.31E+09	
4	2.40E+07	4.72E+07	9.40E+07	1.87E+08	3.75E+08	7.48E+08	
8	1.25E+07	2.45E+07	4.82E+07	9.55E+07	1.91E+08	3.80E+08	7.61E+08
16	6.96E+06	1.31E+07	2.51E+07	4.16E+07	7.61E+07	1.45E+08	2.82E+08
32	4.67E+06	8.05E+06	1.43E+07	2.70E+07	5.23E+07	8.67E+07	1.59E+08

A runtime limit on the size of dynamically-allocable arrays was encountered at 8M entries, and thus measurements are not available for several processor counts on the larger array sizes.

The MPI implementation scales better than any of the UPC implementations. The UPC implementation of the random sample sort is closest in performance, with a 22.7 speed-up for 32 processors on a 2M-entry list. The MPI implementation is an order of magnitude faster than any of the UPC implementations. Only 2 processors are needed for the MPI implementation to beat the sequential implementation in execution time for the larger (2M- and 4M-entry) lists.

Table 5. Scaling of MPI Radix Sort

Processors	List Length				
	131072	262144	524288	2097152	4194304
1	1.00	1.00	1.00	1.00	1.00
2	1.97	1.98	1.98	1.98	1.98
4	3.39	3.44	3.46	3.47	3.47
8	6.49	6.63	6.74	6.81	6.82
16	11.71	12.37	12.94	15.61	17.08
32	17.43	20.20	22.65	24.08	24.83

6.3 Prim’s Algorithm

A version of Prim’s algorithm was also implemented in MPI as a benchmark. The MPI implementation, Prim-MPI, was based on a direct translation of the algorithm without tuning. Prim-MPI showed near-linear speedup on the 10,000-, 40,000-, and 160,000-node random graphs. Its absolute time in processor cycles was consistently one-fourth to one-fifth that of the best UPC implementation for any size graph or number of processors.

We present in Table 6 the results for the 40,000-node graph in tabular form.

Table 6. Run Times for 40,000-Node Graph

<i>Processors</i>	<i>Prim-UPC-1</i>	<i>Prim-UPC-2</i>	<i>Prim-UPC-3</i>	<i>Prim-UPC-4</i>	<i>Prim-MPI</i>
1	1.23E+13	2.43E+12	2.35E+12	2.48E+12	4.86E+11
2	N/A	2.03E+12	1.68E+12	2.01E+12	2.48E+11
4	7.57E+12	1.02E+12	7.01E+11	1.01E+12	1.23E+11
8	4.19E+12	5.14E+11	3.15E+11	3.62E+11	6.18E+10
16	2.48E+12	2.63E+11	1.53E+11	2.62E+11	3.13E+10

7 Future Research

There are several areas of future work in evaluating UPC, and some work would need to be done on the implementation and runtime support to achieve good performance to make UPC competitive with MPI.

7.1 Performance Model

Work was done to develop a predictive performance model for UPC as well as describe the costs of various UPC operations on test systems. The goal of this line of research remains

the development of a set of configuration scripts or other executables that can be executed on a system, measure specific parameters, and then return suggestions to the programmer for a performance-oriented programming style.

7.1.1 Model: V, T, B

A simple first model with overhead (V), transmission cost (T) and barrier time (B) parameters was devised. We assume that communication cost is considerably more than computation cost and thus dominates the runtime of the program. The UPC implementations were then analyzed with these models to develop an equation to describe the behavior of the code. For example, we have for the random sample sort:

- Copy from List to local: $V + (N/P)T$
- Copy samples to Samples array: $V + ST$
- Computation of Pivots: $V + SPT$
- Copy of bucket lengths: $V + PT$
- Computation of global offsets: $V + P^2T$
- Copy back of buckets: $V + (N/P)T$
- Getting new size of chunk: $V + PT$
- Sorting of bucket: $2(V + L(N/P)T)$
- 3 Barrier synchronization: $3B$

which gives a formula of $9V + 3B + (2(1 + L)N/P + S + SP + P + P^2)T$. S is the oversampling factor and L is the maximal fraction of the array that is allocated to a single thread by the selection of the pivots. With a large oversampling factor, this should be reasonably well-balanced. N is the length of the list and P is the number of threads. We have assumed that when a thread starts a request for a large section of a shared array, it pays overhead costs only at the start.

Unfortunately, the above formula is not helpful. The constants were approximated to fit several sections of the code, but these then did not provide a reasonable estimate of the total run time of the code. Thus, a better model is needed.

7.1.2 UPC Barriers

UPC Barriers are very expensive on the test systems. On the Cray, a strange behavior was observed that barrier costs spiked at two threads and then declined as the number of threads increased. This needs further analysis and testing to determine whether this is indeed a real phenomenon or an artifact of the testing strategy adopted.

7.2 Library Function

Some preliminary work has been done to convert portions of Sandia's Zoltan library to UPC. However, there are some impediments to building a UPC library:

- Platform: The test systems used are inappropriate for this development. They are not operated by Sandia and, as noted in the next subsection, do not provide full support for UPC in terms of the current language standard and runtime support to fully utilize hardware resources.
- Functions in UPC: In the Cray implementation, the typing of shared pointers ignores the block size of the striped array. This leads to difficulties when creating a library of functions where ordering in an array matters – such as a sorting functions. It is possible to work around this by passing the block size as an integer parameter to the function as follows:

```
void function(shared int * myArray, int blocking) {  
    shared [blocking] int * newArray;  
    ...  
    newArray = myArray;  
}
```

However, the work around is somewhat awkward.

7.3 Implementation

Two test systems were available for this project: a Cray T3e/750 with 48 nodes and an HP/Compaq cluster with eight nodes, each having four 800-MHz processors. The following shortcomings in the implementations were encountered:

- Node Limitations: On the HP/Compaq, the UPC runtime was restricted to run within a single node only. Furthermore, within the four-processor node, no more than two processors could be used to execute a UPC program. Thus, we could not use the HP/Compaq cluster for scalability testing.

- **Memory Limitations:** Both the Cray and the HP/Compaq suffered from limits on the maximum segment size for shared items. On the HP/Compaq, the largest integer array that could be allocated was 512K entries. On the Cray, this was limited to 16M entries. In both cases, these could only be allocated if no other UPC shared memory was allocated. This limited the size of problem that could be tested, and ruled out all testing on the HP/Compaq.
- **Full Standard Support:** The compiler on the Cray lacked full support for several language features. For example, in allocation of shared items dynamically, the compiler only supported the `upc_all_alloc` collective function. Neither the `upc_global_alloc`, which allocates shared memory across all threads and is called by a single thread, nor the `upc_local_alloc`, which allocates shared memory only on the calling thread, were supported. The UPC string functions – such as `upc_mem_cpy` – also were not supported.
- **Caching:** Tests suggest that no caching was used with UPC shared items on the Cray. Although the UPC implementation on the Cray uses hardware to handle shared memory requests, and, as we have seen in Section 6, is thus considerably faster in accessing a single item from shared memory than an MPI transmission, the locality at the heart of the UPC specification is somewhat lost without the caching. Although UPC code is written to take advantage of locality through thread affinity, the lack of caching makes this a moot point on the Cray. A version of the radix sort that took pains to use much more locality than that code in the appendix was developed using our macros (Section 5) and this led to a slight slow-down in execution time.
- **No Collective Operations:** The UPC implementation available on the Cray lacked collective operations for gather and scatter operations. As we saw in section 6, the final UPC radix sort implementation could be readily translated to MPI, using only MPI collective operations. Further, although to access a single integer in MPI using a send/receive combination takes an order of magnitude more time than to access a single integer in UPC shared space, the MPI radix sort ran an order of magnitude faster than the UPC code. This suggests that efficient collective operations in UPC could dramatically improve the performance of UPC code.

We thus have several clear requirements of a UPC compiler and runtime if UPC is to be used in a production environment at Sandia:

- We should be able to deploy UPC code to all processing elements in a cluster, not just nodes on a single shared-memory processor board. If a job uses 500 nodes in MPI, we should expect to be able to dedicate 500 nodes to a single UPC instance of the same job.
- The size of the shared memory segment should be substantially larger, so that large problems can be solved with UPC code. Ideally, the shared memory segment size should only be limited by the physical memory of the processing elements.

- The compiler and runtime should support the current UPC specification as well as collective operations, and these should be tuned for the hardware. Fast collective operations can be implemented while still remaining relatively unobtrusive extensions to C.
- Locality, as presented to the programmer through the UPC extensions, should in fact be realized in hardware. On the Cray, although the programmer can see locality in arrays through thread affinity, this is not borne out by the implementation and hardware. What may be needed is a third memory model, in addition to the **strict** and **relaxed** already used in UPC, that would be programmer-directed. The programmer could, for performance gains, issue a pragma to the compiler to disable synchronization of specific shared items until the next synchronization event (barrier or notify/wait). Then, the compiler would be free to optimize those memory accesses as private memory up to the synchronization event. This could be realized by copying the shared items to private memory and only using the private memory addresses within the code up to the synchronization.

UPC offers the application programmer the option of putting the burden on the compiler with `upc_forall` and similar directives, or of tuning the code to exploit locality where possible. The programmer should be provided software tools that allow this code tuning to be realized into real performance gains to tackle large problems. For this, an improved compiler and runtime are needed.

7.4 Petaflop Architecture

One of the initial goals of the UPC project at Sandia was to determine whether the GAS model in general and UPC in particular would be an appropriate programming environment for petaflop-scale computing. This continues to be an open topic, especially as new and different petaflop architecture ideas are considered. Lacking access to petaflop architecture candidate designs, no work was done on this.

8 Conclusions

UPC and the GAS programming model remain promising new ideas for Sandia's future computing needs, but are not yet ready for deployment in an environment where performance is key. Our sample codes for sorting and minimum spanning tree applications were slower in overall execution time than MPI counterparts. UPC compilers and runtimes are available for select systems, and these have varying degree of support for language features. However, there are several important points that we consider provide hope for future use of UPC or similar GAS languages:

- **Immaturity of Implementation:** The UPC specification dates from 1999, with the first fully-featured specification accepted in 2003. The test systems had what could be termed immature implementations, and many of our complaints stem from problems with these implementations. Limitations on shared memory segment size, number of processors that could be utilized, and lack of support for thread affinity and locality at the core of UPC were all flaws encountered with the implementations available on the Cray and HP/Compaq. Better runtime and compiler support, as outlined in Section 7.3, are needed to improve the viability of the language and improve performance.
- **Scaling:** As shown with the random sample sort and the Prim-UPC-3 implementations, good scaling can be achieved with UPC code. The performance of the Prim-UPC-3 code, and why better load balancing improved its scaling, still needs to be analyzed, as the MPI code was implemented without load balancing and yet achieved good scaling on the same test graphs.
- **Ease of Programming:** As described in Sections 2.4 and 3.1, UPC development has the advantages of shared memory development. A functional, correct program can be implemented, and then iterative improvements can be made to achieve performance goals. The language extensions are simple and unobtrusive extensions to standard C.
- **New Language Features:** The MPI radix sort scaled well using MPI collective operations for scatter, gather, and broadcast. Although our measurements show that, on the Cray test system, UPC shared memory accesses of single items are an order of magnitude faster than MPI send/receive communications, the UPC code was overall an order of magnitude slower than the MPI code. This suggests that good, well-tuned collective operations can greatly improve performance. No bulk transfer operations or collective operations were available on the test system; their availability may well lead to better performance.

Work needs to be done on compiler and runtime support for UPC before it can be deployed in a production environment where performance and scalability is key. However, new language features and improvements to the UPC programming environment may make this a viable option in the future.

9 Acknowledgements

We wish to thank Quentin F. Stout of the University of Michigan for his assistance with this work. We also would like to thank Ronald Brightwell and Sue Goudy for assistance with this project and an initial poster presentation at SC03. We would like to express our appreciation for many constructive comments from Mahesh Rajan. This work was supported in part by Sandia Contract 235451, a Sandia National Laboratories summer internship, and a Department of Homeland Security fellowship.

References

- [1] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Gregory Plaxton, Stephen J. Smith, and Marco Zagha, *A comparison of sorting algorithms for the connection machine cm-2*, ACM Symposium on the Theory of Computation, 1991, pp. 3–16.
- [2] Sébastien Chauvin, Proshanta Saha, Francois Cantonnet, Smita Annareddy, and Tarek El-Ghazawi, *Upc manual*, 2003.
- [3] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper, *Upc language specification v 1.1.1.*, 2003.
- [4] David Eppstein, *Minimum spanning trees*, ICS 161 Lecture Notes, David Eppstein, 1996.
- [5] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to parallel computing*, second ed., Addison Wesley, 2003.
- [6] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach*, third ed., Morgan Kaufman, 2003.
- [7] R. Numrich and J. Reid, *Co-array fortran for parallel programming*, 1998.
- [8] Robert Sedgewick, *Algorithms in c, parts 1-4: Fundamentals, data structures, sorting, searching*, third ed., Addison Wesley Professional, 1998.
- [9] UPC Consortium, *Upc collective operations specification v1.0*, 2003.

A Comparative Sorting Algorithm Results

Figure A.1. (a) Speed-Up for Sorting a 128K-Entry List Good speed-up was seen with the random sample sort. The radix sort showed reasonable speed-up until 32 processors. The parallel bubble and bottom-up merge sorts showed anemic scaling, and the merge with Batcher's odd-even operators showed almost no scaling. No scaling is reported for QuickSort as the UPC version required two processors. The best UPC scaling was seen with the UPC implementation of sample sort, which is competitive with that of the MPI radix sort. **(b) Sorting Time in Processor Cycles for 128K-Entry List** The fine-grained sorts, QuickSort and Merge with Batcher's Odd-Even operators, performed poorly, almost an order of magnitude slower than the other four sorts.

Figure A.2. (a) Speed-Up for Sorting a 512K-Entry List Good speed-up was seen with the random sample sort. The radix sort showed reasonable speed-up until 32 processors. The parallel bubble and bottom-up merge sorts showed anemic scaling, and the merge with Batcher’s odd-even operators showed almost no scaling. No scaling is reported for QuickSort as the UPC version required two processors. The best UPC scaling was seen with the UPC implementation of sample sort, which is competitive with that of the MPI radix sort. **(b) Sorting Time in Processor Cycles for 512K-Entry List** The fine-grained sorts, QuickSort and Merge with Batcher’s Odd-Even operators, performed poorly, almost an order of magnitude slower than the other four sorts. No further numbers will be reported for these as they either ran out of shared memory or did not complete after several weeks on larger problem sizes.

Figure A.3. (a) Speed-Up for Sorting 2M-Entry List We see anemic scaling for the parallel bubble and bottom-up merge sorts, reasonably good scaling from the radix sort, and good scaling from the random sample sort. The best UPC scaling was seen with the UPC implementation of sample sort, which is competitive with that of the MPI radix sort. **(b) Sorting Time in Processor Cycles for 2M-Entry List** The radix and random sample sorts are considerably faster than the bottom-up merge or the parallel bubble sorts. The radix sort is the fastest sort, but the superior scaling of the random sample sort leads to almost the same time for 32 processors. The MPI radix sort is consistently an order of magnitude faster than any of the UPC implementations.

Figure A.4. (a) Speed-Up for Sorting 4M-Entry List Scaling is only shown for the radix sort, as it is the only sort that completed for the UPC uniprocessor case. Radix exhibits good scaling up to 16 processors, but appears to level off quickly at 32. The UPC radix implementation, however, lags the MPI implementation. **(b) Sorting Times in Processor Cycles for 4M-Entry List** The radix and random sample sorts are considerably faster than the bottom-up merge or the parallel bubble sorts. The radix sort is the fastest sort, but the superior scaling of the random sample sort leads to almost the same time for 32 processors. The MPI radix sort is consistently an order of magnitude faster than any of the UPC implementations.

Figure A.5. Sorting Times in Processor Cycles for 8M-Entry List Only the parallel bubble, radix, and random samples sorts were sufficiently efficient with shared memory resources to complete sorting the 8M-entry list. The radix sort is still the fastest sort, but the superior scaling of the random sample sort leads to almost the same time for 32 processors. The bubble sort is significantly slower than the other two.

B Summary of Sorting Results By Algorithm

The following is a breakdown of results per algorithm for each of the UPC sorting algorithm implementations.

Figure B.6. (a) **Sorting Time for Parallel Bubble Sort** The parallel bubble sort did not scale well beyond a few processors, as can be seen by the relative similarity of slope of the 8, 16, and 32 processor lines. (b) **Bubble Sort List Length Scaling from 128K-Entry** The processing time for the parallel bubble sort scaled linearly with the increasing list length, irrespective of the number of processors. Scaling is measured with respect to a 128K-entry list.

Figure B.7. (a) Sorting Time for Bottom-Up Merge Sort The bottom-up merge sort did not scale well beyond a few processors, as can be seen by the relative similarity of slope of the 8, 16, and 32 processor lines. **(b) Merge Sort List Length Scaling from 128K-Entry** The processing time for the bottom-up merge sort scaled linearly with the increasing list length, irrespective of the number of processors. Scaling is measured with respect to a 128K-entry list.

Figure B.8. (a) Sorting Time for Radix Sort The radix sort did not exhibit the same scaling as the random sample sort for increasing the number of processors, but still exhibited reasonably good scaling. The execution times decrease as the number of processors increases. Observe that the graph for 32 processors is sublinear in form. **(b) Radix Sort List Length Scaling from 128K-Entry** The processing time for the Radix sort algorithm scaled linearly with the size of the list to be sorted for up to 8 processors. However, we observe that for 16 and 32 processors, we have sublinear growth. This leads us to conclude that this is a good candidate for a general purpose UPC sort algorithm. Scaling is measured with respect to a 128K-entry list.

Figure B.9. (a) Sorting Time for Sample Sort The random sample sort exhibited good scaling as the number of processors increased, as shown with the increasingly shallow execution time lines. **(b) Sample Sort List Length Scaling from 128K-Entry** The processing time for the random sample sort algorithm scaled linearly with the size of the list to be sorted irrespective of number of processors, except for a slight jump for four processors and a 8M-entry list. Scaling is measured with respect to a 128K-entry list.

C Minimum Spanning Tree Results

Figure C.10. Time in processor cycles for each implementation of Prim's algorithm – Prim-UPC-1, Prim-UPC-2, Prim-UPC-3, Prim-UPC-4, and Prim-MPI – for varying numbers of processors and a 10,000-node graph. The graph was constructed using the sieve algorithm and $p = 1$. The Prim-MPI implementation was consistently faster than any of the UPC implementations, by a factor of at least 4. Prim-UPC-3, with oblivious load balancing, was the fastest UPC implementation.

Figure C.11. Speedup for each implementation of Prim's algorithm for a 10,000-node graph. A linear speedup line is provided for reference. Near-linear speedup was achieved by Prim-UPC-3 and Prim-MPI. Good speedup was achieved by Prim-UPC-2 and Prim-UPC-4. The naive UPC implementation, Prim-UPC-1, did not exhibit good scaling.

Figure C.12. Time in processor cycles for each implementation of Prim’s algorithm – Prim-UPC-1, Prim-UPC-2, Prim-UPC-3, Prim-UPC-4, and Prim-MPI – for varying numbers of processors and a 40,000-node graph. The graph was constructed using the sieve algorithm and $p = 1$. The Prim-MPI implementation was consistently faster than any of the UPC implementations, by a factor of at least 4. Prim-UPC-3, with oblivious load balancing, was the fastest UPC implementation for any more than a single processor.

Figure C.13. Speedup for each implementation of Prim's algorithm for a 40,000-node graph. A linear speedup line is provided for reference. Near-linear speedup was achieved by Prim-UPC-3 and Prim-MPI. Good speedup was achieved by Prim-UPC-2 and Prim-UPC-4, with an bump in speedup for eight processors with Prim-UPC-4 most likely be due to underlying graph properties (i.e., the division for 8 processors stayed balanced). The naive UPC implementation, Prim-UPC-1, did not exhibit good scaling.

Figure C.14. Time in processor cycles for each implementation of Prim's algorithm – Prim-UPC-1, Prim-UPC-2, Prim-UPC-3, Prim-UPC-4, and Prim-MPI – for varying numbers of processors and a 160,000-node graph. The graph was constructed using the sieve algorithm and $p = 1$. The Prim-MPI implementation was consistently faster than any of the UPC implementations, by a factor of at least 4. Prim-UPC-3, with oblivious load balancing, was the fastest UPC implementation.

Figure C.15. Speedup for each implementation of Prim's algorithm for a 160,000-node graph. A linear speedup line is provided for reference. Near-linear speedup was achieved by Prim-UPC-3 and Prim-MPI. Good speedup was achieved by Prim-UPC-2 and Prim-UPC-4. The naive UPC implementation, Prim-UPC-1, did not exhibit good scaling.

D Communication Latency and Cost Measurements

Figure D.16. Average Cost per Element for Writes to a UPC Array. Average per processor are 20% trimmed means of average cost per element for accessing arrays of sizes ranging from 1024 (2^{10}) to 4194304 (2^{22}) elements. Little variation was observed as the array size varied. The “Thread 0” line is for the test in which only thread 0 writes to the array; the “All Threads” line is for the test in which all threads write to the array simultaneously. The cost per array entry varies little for write operations as the number of processors increases.

Figure D.17. Access time per array element for reads from a large shared array. Average costs from Table 2 are graphed for between 1 and 32 processors (Experimental line). Equation 1 is graphed with constants $\beta = 197.66$, the experimental cost per access for a single UPC thread, and $\alpha = 518.9$, the average of the values computed from the experimental data (Constants line). Using a fixed constant for α does not fit the experimental data well when P increases. A third line (Fit Curve line) is graphed using equation 1 but so that α satisfies $P = 0.0053\alpha^2 - 5.0685\alpha + 1211.6$, an equation that fits the experimental data with high correlation ($R^2 = 0.9979$). This suggests that $\alpha(P) = \alpha' + \Theta(\sqrt{P})$, where α' is a constant and $\Theta\sqrt{P}$ term is small. This agrees with the topology of the test system – as it is a three-dimensional taurus, allocation of the processing elements to minimize the maximal distance between processing elements would lead to a packing with edges of size proportional to $\sqrt{P}$ or $\sqrt[3]{P}$.

Figure D.18. Average Cost per Element for Reads from a UPC Array. Average per processor are 20% trimmed means of average cost per element for accessing arrays of sizes ranging from 1024 (2^{10}) to 4194304 (2^{22}) elements. We observe that access to shared memory with local affinity is very expensive (on the order of 200 cycles per array element) and, depending on the number of processors and activity on the system, can cost upwards of 600 cycles per element for remote shared accesses. The curves follow the model from Equation 1. The “Thread 0” line is for the test in which only thread 0 reads from the array; the “All Threads” line is for the test in which all threads read from the array simultaneously.

Figure D.19. Average round-trip time for transfer of a large integer array using MPI with acknowledgement. A large packet was transferred from processor 0 to each of the other processors, and each processor returned a small acknowledgement packet. Average time over the other 31 processors is shown. There was very small variance (often with a standard deviation considerably less than 10% of the average time), with no discernable trend from architectural features as was observed with the UPC test. Equation *A* is an approximation with $C = 5.08 \cdot N + 3751.2$, where C is the average cost and N is the number of integers in the payload. This corresponds to the “short” protocol. Equation *B* is an approximation with $C = 1.89 \cdot N + 203711$, which corresponds to the “long” protocol. *A* is a good fit for payloads of up to one million entries (2^{20}), whereas *B* is a good fit for larger payloads.

Figure D.20. Average cost for executing an `MPI_Allgather` collective operation. Using 32 processors, up to four million integers (2^{22}) items were exchanged. For larger quantities exchanged, cost fits the equation $C = 8.85 \cdot N + 189562$, where C is the average cost and N is the total number of integers moved.

Figure D.21. Access time per array element for varying array strides on T3E. We observe that an access to memory outside the cache on the test system consumes roughly 57 cycles. This compares to the write to shared on UPC with optimization taking between 85 – 90 cycles.

E Code Samples

E.1 Sorting Algorithms

E.1.1 Testing Harness

This testing harness was used for all UPC sorting algorithm tests. The function call varied – that for the random sample sort is shown.

```
#include <stdio.h>
#include <stdlib.h>
#include <upc_relaxed.h>
#include <sys/types.h>
#include <time.h>
#include <sys/times.h>
#include <limits.h>

shared int List[N];

#include "sample2.h"

int main() {

    FILE * number_file;
    int i, j;
    char line[100];
    struct tms timerStructure;
    clock_t start, end;

    if (MYTHREAD == 0) {
        // fill the list from the random number file
        number_file = fopen("Random.txt", "r");

        if (number_file == NULL) {
            fprintf(stderr, "Could not read random number file.\n");
            exit(0);
        }

        for (i = 0; i < N; i++) {
            fgets(line, sizeof(line), number_file);
            sscanf(line, "%d", &j);
            List[i] = j;
        }
    }
```

```

        fclose(number_file);
    }

    if (MYTHREAD == 0) {
        start = times(&timerStructure);
    }

    upc_barrier;

    // sort the list
    SampleSort();

    upc_barrier;

    if (MYTHREAD == 0) {
        end = times(&timerStructure);
        printf("%d\n", end - start);
    }
}

```

E.1.2 QuickSort

The following is a recursive QuickSort with very fine-grained parallelism. Thread groups – subsets of threads that could be independently synchronized – might make such a recursive formulation, where threads advance down a recursion tree, splitting off on different branches into subsets, feasible. At present, this performs poorly.

```

#include <upc_relaxed.h>
#include "local_quicksort.h"
#include "prefix.h"
#include "thread_concat_target.h"

int ComputePivot(int * A, int length) {
    int i, j, k;
    int localPivot[3 * THREADS];
    shared int * PivotArray;
    shared int * result;

    PivotArray = (shared int *) upc_all_alloc(sizeof(shared int), 3 * THREADS);
    result = (shared int *) upc_all_alloc(sizeof(shared int), 1);
}

```

```

PivotArray[3 * MYTHREAD] = A[0];
PivotArray[3 * MYTHREAD + 1] = A[length/2];
PivotArray[3 * MYTHREAD + 2] = A[length - 1];

upc_barrier;

if (MYTHREAD == 0) {
    for (i = 0; i < 3 * THREADS; i++) {
        localPivot[i] = PivotArray[i];
    }

    QSort(localPivot, 0, 3 * THREADS);

    (*result) = localPivot[3 * THREADS / 2];
}

upc_barrier;

return (*result);
}

void QuickSort(int left, int right) {
    int temp[THREADS], A[(right - left)/THREADS + 1],
        B[(right - left)/THREADS + 1], lowTail, highHead;
    int i, j, k, pivot;
    shared int * prefixes;
    int length = (right - left) / THREADS;

    if (MYTHREAD < (right - left - length * THREADS)) length++;

    if (right - left <= THREADS) {
        if (MYTHREAD == 0) {
            for (i = 0; i < right - left; i++) {
                temp[i] = List[left + i];
            }

            QSort(&temp[0], 0, right - left);

            for (i = 0; i < right - left; i++) {
                List[left + i] = temp[i];
            }
        }
        upc_barrier;
    }
}

```

```

        return;
    }

    if (left >= right - 1) return;

    for ( j = 0; j < length; j++) {
        A[j] = List[MYTHREAD + j * THREADS + left];
    }

    pivot = ComputePivot(A, length);

    for (i = 0, lowTail = 0, highHead = length - 1; i < length; i++) {
        if (A[i] < pivot) {
            B[lowTail] = A[i];
            lowTail++;
        } else {
            B[highHead] = A[i];
            highHead--;
        }
    }

    prefixes = ThreadPrefix(lowTail);

    upc_barrier;

    ThreadConcatenate(List + left, B, lowTail);
    ThreadConcatenate(List + left + prefixes[THREADS - 1],
        B + lowTail, length - lowTail);

    upc_barrier;

    QuickSort(left, left + prefixes[THREADS - 1]);
    QuickSort(left + prefixes[THREADS - 1], right);

}

```

E.1.3 Odd-Even

This is a fine-grained merge sort that uses the UPC parallel loop directive, `upc_forall`, and Batcher's Odd-Even operators . The nesting of the UPC loop directive was varied to little effect on performance.


```

#include <upc_relaxed.h>

#ifndef ODDEVEN
#define ODDEVEN

void OddEvenMerge(shared int * a, int l, int m, int r) {
    int i, j, k, temp;
    int span = r - l + 1;

    for (k = span / 2; k > 0; k /= 2) {
        for (j = k % (span / 2); j + k < span; j += k + k) {
            upc_forall (i = 0; i < k; i++; &a[l + j + i]) {
                if (a[l+j+i] > a[l+j+i+k]) {
                    temp = a[l+j+i];
                    a[l+j+i] = a[l+j+i+k];
                    a[l+j+i+k] = temp;
                }
            }
            upc_barrier;
        }
    }
}

void OddEven(shared int * a, int l, int r) {
    int m, i;
    int temp;

    for (m = 1; m <= r - l; m = m + m) {
        for (i = l; i <= r - m; i += m + m) {
            temp = (i + m + m - 1 < r) ? i + m + m - 1 : r;
            OddEvenMerge(a, i, i + m - 1, temp);
        }
    }
}

#endif

```

E.1.4 Bottom-up Merge

This references a local radix sort, which is also referenced by the parallel bubble sort and the random sample sort.

```

#include <upc_relaxed.h>
#include "local_radix.h"

#ifndef PARALLELMERGE
#define PARALLELMERGE

void Merge(int * T, int * A, int * B, int la, int lb) {
    int tempTarg[la + lb];
    int temp = 0, ptra = 0, ptrb = 0;

    while (ptra < la && ptrb < lb) {
        if (A[ptra] < B[ptrb]) {
            tempTarg[temp] = A[ptra];
            ptra++;
        } else {
            tempTarg[temp] = B[ptrb];
            ptrb++;
        }
        temp++;
    }

    while (ptra < la) {
        tempTarg[temp] = A[ptra];
        temp++;
        ptra++;
    }

    while (ptrb < lb) {
        tempTarg[temp] = B[ptrb];
        temp++;
        ptrb++;
    }

    for (temp = 0; temp < ptra + ptrb; temp++) {
        T[temp] = tempTarg[temp];
    }
}

void MergeSort() {
    int i, list_length, start_index, ordering;
    int local_array[N], temp_list[N];
    int j, k, m, n, p, q, r;

```

```

// copy the list
for (i = 0; i < N / THREADS; i++) {
    temp_list[i] = List[MYTHREAD * N / THREADS + i];
}

// sort your piece of the array
RSort(temp_list, N / THREADS, 16);

// copy your list back!
for (i = 0; i < N / THREADS; i++) {
    List[MYTHREAD * N / THREADS + i] = temp_list[i];
}

upc_barrier;

// now we need to merge these things together
for (i = 2; i <= 2*THREADS-1; i <= 1) {
    if (MYTHREAD % i == 0) {
        if ((MYTHREAD+i/2) * N / THREADS >= N)
            ;
        else {
            if ((MYTHREAD + i) * N / THREADS >= N)
                k = N - (i/2 + MYTHREAD) * N / THREADS;
            else
                k = i/2 * N / THREADS;

            // copy the new half
            for (j = 0; j < k; j++) {
                temp_list[i/2 * N / THREADS + j] =
                    List[(MYTHREAD + i/2) * N / THREADS + j];
            }

            // merge them
            Merge(&temp_list[0],
                &temp_list[0],
                &temp_list[i/2 * N / THREADS],
                i/2 * N / THREADS,
                k);

            // move them back
            k += i/2 * N / THREADS;

```

```

        for (j = 0; j < k; j++) {
            List[MYTHREAD * N / THREADS + j] = temp_list[j];
        }
    }

    // we unfortunately have to synchronize again
    upc_barrier;
}

}

#endif

```

E.1.5 Parallel Bubble

This references a local radix sort, which is also referenced by the bottom-up merge sort and the random sample sort.

```

#include <upc_relaxed.h>
#include "local_radix.h"

/* Function: MergeLowerHalf
 * Purpose: Given two lists of integers, merge them halfway and put
 * the sorted list of the lesser half into the target array
 */
void MergeLowerHalf(int * T, int * A, int * B, int length) {
    int TempTarg[length];
    int tempPtr = 0, aPtr = 0, bPtr = 0;

    /* proceed through the length of A and B, which */
    /* is 1/2 the total length */
    for (tempPtr = 0; tempPtr < length; tempPtr++) {
        if (A[aPtr] < B[bPtr]) {
            TempTarg[tempPtr] = A[aPtr];
            aPtr++;
        } else {
            TempTarg[tempPtr] = B[bPtr];
            bPtr++;
        }
    }

    /* Copy from a temporary array to the target in case T = A or B */

```

```

        for (tempPtr = 0; tempPtr < length; tempPtr++) {
            T[tempPtr] = TempTarg[tempPtr];
        }
    }

/* Function: MergeUpperHalf
 * Purpose: Given two lists of integers, merge them halfway and put
 *          the sorted list of the upper half into the target array
 */
void MergeUpperHalf(int * T, int * A, int * B, int length) {
    int TempTarg[length];
    int tempPtr = length - 1, aPtr = length - 1, bPtr = length - 1;

    /* do the merge backwards to get the upper half */
    for (tempPtr = length - 1; tempPtr >= 0; tempPtr--) {
        if (A[aPtr] > B[bPtr]) {
            TempTarg[tempPtr] = A[aPtr];
            aPtr--;
        } else {
            TempTarg[tempPtr] = B[bPtr];
            bPtr--;
        }
    }

    /* copy from temp in case T = A or B */
    for (tempPtr = 0; tempPtr < length; tempPtr++) {
        T[tempPtr] = TempTarg[tempPtr];
    }
}

/* we use these instead of List for faster access */
shared int Arrays[THREADS][N / THREADS];

/* Function: Bubble()
 * Purpose: Parallel version of bubble sort
 *
 * Note that this expects the numbers to start in List with length N
 */
void Bubble() {
    int length = N;
    int i, j, k, temp[length/THREADS], temp2[length/THREADS];

    /* move from list to Bubble's arrays */
    for (i = 0; i < length / THREADS; i++) {

```

```

    Arrays[MYTHREAD][i] = List[MYTHREAD + i * THREADS];
}

/* copy out of the arrays into a local variable */
/* in retrospect, this and the above could be merged */
for (i = 0; i < length / THREADS; i++) {
    temp[i] = Arrays[MYTHREAD][i];
}

/* do a local radix sort */
RSort(&temp[0], length/THREADS, 16);

/* move from the local array back to our shared arrays */
for (i = 0; i < length / THREADS; i++) {
    Arrays[MYTHREAD][i] = temp[i];
}

upc_barrier;

/* now, we have to bubble our lists */
for (i = 0; i < THREADS; i++) {

    /* copy from the shared memory to a local temporary variable */
    for (j = 0; j < length / THREADS; j++) {
        temp[j] = Arrays[MYTHREAD][j];
    }

    /* if parity of thread matches parity of loop iteration, */
    /* then that thread does a lower half merge */
    if ((MYTHREAD - i) % 2 == 0) {
        if (MYTHREAD != THREADS - 1) {
            /* copy the shared arrays */
            for (j = 0; j < length / THREADS; j++) {
                temp2[j] = Arrays[MYTHREAD + 1][j];
            }

            /* do the merge */
            MergeLowerHalf(&temp[0], &temp[0], &temp2[0], length/THREADS);
        }
    } else {
        /* when parity mismatches, do an upper half merge */
        if (MYTHREAD != 0) {
            /* copy the shared arrays */
            for (j = 0; j < length / THREADS; j++) {

```

```

        temp2[j] = Arrays[MYTHREAD - 1][j];
    }

    /* do the merge */
    MergeUpperHalf(&temp[0], &temp[0], &temp2[0], length/THREADS);

    }
}

upc_barrier;
for (j = 0; j < length / THREADS; j++) {
    Arrays[MYTHREAD][j] = temp[j];
}

upc_barrier;
}

/* copy out of our arrays back to list */
k = MYTHREAD * length / THREADS;
for (i = 0; i < length / THREADS; i++) {
    List[i + k] = Arrays[MYTHREAD][i];
}

upc_barrier;
}

```

E.1.6 Radix

The following version of the radix sort code for UPC shows several of the final design ideas. The `OVERDEC` macro was used to vary how much over-decomposition of the data was used to smooth wide-ranging execution times for small radix values that led to initial super-linear speed-up. Bucketing was inlined to allow for a combination of local bucketing and update of global counts in the `BucketLengths` array. `TargetOffsets`, originally a shared array used for a collective parallel-prefix computation of copy-back locations, is now a private array, and each thread computes the parallel prefix itself, saving on synchronization. Originally, a “copy-in-place” strategy was attempted, whereby, except for the last iteration, the buckets were copied back to the thread that would need them on the next iteration rather than just into the list in list order. This added overhead and led to an increase in execution time over the two-copy implementation.

```

#include <upc_relaxed.h>
#include "prefix.h"

```

```

#include <math.h>

#ifndef OVERDEC
#define OVERDEC 256
#endif

#ifndef SHARED_RADIX
#define SHARED_RADIX

shared int * BucketLengths;

shared int depth;

void RadixSort() {
    int A[N/THREADS];
    int * TargetOffsets;
    int B[N/THREADS];
    int ** Buckets;
    int * Counts;
    int i, j, k, max = 0;
    shared int * maximums;
    double logResult;
    int radix, multiple = 1;

    //radix = (THREADS == 1) ? 2 : THREADS;
    radix = THREADS * OVERDEC;

    BucketLengths = (shared int *) upc_all_alloc(sizeof(shared int),
        radix * THREADS);
    TargetOffsets = (int *) malloc((sizeof(int) * radix * THREADS));
    Buckets = (int **) malloc(sizeof(int *) * radix);
    Counts = (int *) malloc(sizeof(int) * radix);

    // partition Data
    for (i = 0; i < N / THREADS; i++) {
        A[i] = List[MYTHREAD + i * THREADS];
        if (A[i] > max) max = A[i];
    }

    // compute depth
    maximums = ThreadPrefixExt(max, &maximum);

    if (MYTHREAD == 0) {

```



```

    logResult = log(maximums[THREADS - 1]) / log(radix);

    if (logResult > (int) logResult) {
        depth = (int) logResult + 1;
    } else {
        depth = (int) logResult;
    }
}

upc_barrier;

for (j = 0; j < depth; j++) {
    /* other than the first time, we need a fresh bucket */
    if (j != 0) {
        for (i = 0; i < N / THREADS; i++) {
            A[i] = List[i + MYTHREAD * N / THREADS];
        }
    }
    for (i = 0; i < radix; i++) {
        Counts[i] = 0;
    }

    /* find the count of each */
    for (i = 0; i < N / THREADS; i++) {
        k = (A[i] / multiple) % radix;
        Counts[k]++;
    }

    /* do a mini-prefix */
    Buckets[0] = &B[0];
    BucketLengths[MYTHREAD] = Counts[0];
    for (i = 1; i < radix; i++) {
        Buckets[i] = Buckets[i - 1] + Counts[i - 1];
        BucketLengths[i * THREADS + MYTHREAD] = Counts[i];
        Counts[i - 1] = 0;
    }
    Counts[radix - 1] = 0;

    /* now, put into a bucket */
    for (i = 0; i < N / THREADS; i++) {

```

```

        k = (A[i] / multiple) % radix;
        Buckets[k][Counts[k]] = A[i];
        Counts[k]++;
    }

    upc_barrier;

    TargetOffsets[0] = 0;
    for (i = 1; i < radix * THREADS; i++) {
        TargetOffsets[i] = TargetOffsets[i - 1] + BucketLengths[i - 1];
    }

    /* copy your bucket back */
    for (i = 0; i < radix; i++) {
        for (k = 0; k < Counts[i]; k++) {
            List[TargetOffsets[i * THREADS + MYTHREAD] + k] = Buckets[i][k];
        }
    }

    multiple *= radix;

    upc_barrier;

}

}

#endif

```

Alternatively, the depth code can be written with one fewer synchronization:

```

// partition Data
for (i = 0; i < N / THREADS; i++) {
    A[i] = List[MYTHREAD + i * THREADS];
    if (A[i] > max) max = A[i];
}

// compute depth
maximums = ThreadPrefixExt(max, &maximum);

logResult = log(maximums[THREADS - 1]) / log(radix);

```

```

        if (logResult > (int) logResult) {
            depth = (int) logResult + 1;
        } else {
            depth = (int) logResult;
        }

upc_barrier;

```

The greatest performance boosts were seen when improving the main loop. By contrast, no real gain was achieved by this modification.

E.1.7 Random Sample

This code is based on that in [1].

```

#include <upc_relaxed.h>
#include "bucketing.h"
#include "local_quicksort.h"
#include "thread_concat_target.h"
#include "local_radix.h"

#ifndef SAMPLE_SORT
#define SAMPLE_SORT

#ifndef OVERSAMPLE
#define OVERSAMPLE 2 * THREADS
#endif

int * ChooseK(int * Src, int length, int k) {
    int i, step;
    int * temp = (int *) malloc(sizeof(int) * k);

    step = (length / (k + 1) < 1) ? 1 : length / (k + 1);
    for (i = 0; i < k; i++) {
        temp[i] = Src[step * i];
    }

    return temp;
}

```

```

int localPivots[THREADS - 1];
shared int Pivots[THREADS - 1];
shared [OVERSAMPLE] int Samples[OVERSAMPLE * THREADS];

int aBsearch(int * A, int x, int L, int H) {
    int M;

    if (L >= H) {
        if (x > A[H]) {
            return H + 1;
        } else {
            return H;
        }
    }

    M = (L + H) / 2;

    if (x > A[M]) {
        return aBsearch(A, x, M + 1, H);
    } else {
        return aBsearch(A, x, L, M - 1);
    }
}

int getPosition(int a) {

    return aBsearch(localPivots, a, 0, THREADS - 2);

}

shared int * Buckets[THREADS * THREADS];
shared int BucketLengths[THREADS * THREADS];
shared int BucketOffsets[THREADS * THREADS];

void SampleSort() {
    int temp[THREADS], A[N / THREADS], B[N / THREADS];
    int * localBuckets[THREADS];
    int * localSamples;
    int localCounts[THREADS];
    int i, j, k;
    int * localData, size;

    // partition List

```

```

for (i = 0; i < N / THREADS; i++) {
    A[i] = List[MYTHREAD + i * THREADS];
}

// Find samples and set up partitions
localSamples = ChooseK(A, N / THREADS, OVERSAMPLE);

for (i = 0; i < OVERSAMPLE; i++) {
    Samples[MYTHREAD * OVERSAMPLE + i] = localSamples[i];
}

upc_barrier;

if (MYTHREAD == 0) {
    for (i = 0; i < OVERSAMPLE * THREADS; i++) {
        B[i] = Samples[i];
    }

    QSort(&B[0], 0, OVERSAMPLE * THREADS);

    for (i = 0; i < THREADS - 1; i++) {
        Pivots[i] = B[(i + 1) * OVERSAMPLE];
    }
}

upc_barrier;

for (i = 0; i < THREADS - 1; i++) {
    localPivots[i] = Pivots[i];
}

// do the bucketing
Bucketing(A, B, localBuckets, N / THREADS, localCounts, THREADS, &getPosition);

// move out of localbucket into a shared array
// this could be further parallelized, but we're only going to 32 proc.
for (i = 0; i < THREADS; i++) {
    BucketLengths[i * THREADS + MYTHREAD] = localCounts[i];
}

upc_barrier;

```

```

if (MYTHREAD == 0) {
    BucketOffsets[0] = 0;
    for (i = 1; i < THREADS * THREADS; i++) {
        BucketOffsets[i] = BucketOffsets[i - 1] + BucketLengths[i - 1];
    }
}

upc_barrier;

for (i = 0; i < THREADS; i++) {
    for (j = 0; j < localCounts[i]; j++) {
        *(List + BucketOffsets[i * THREADS + MYTHREAD] + j) =
            *(localBuckets[i] + j);
    }
}

upc_barrier;

// gather your buckets and do a local sort
size = 0;
for (j = 0; j < THREADS; j++) {
    size += BucketLengths[MYTHREAD * THREADS + j];
}

localData = (int *) malloc(sizeof(int) * size);
i = 0;
for (j = 0; j < THREADS; j++) {
    for (k = 0; k < BucketLengths[MYTHREAD * THREADS + j]; k++, i++) {
        localData[i] = List[BucketOffsets[MYTHREAD * THREADS + j] + k];
    }
}

RSort(localData, size, 16);

// bring it all together in one array
ThreadConcatenate(List, localData, size);
}

#endif

```

E.1.8 MPI Implementation of Radix Sort

The MPI implementation of the radix sort was an almost direct translation of the UPC radix implementation. MPI collective operations were used extensively. The major difficulty was in achieving the load balancing that came for free with UPC. This was done with extensive matrix manipulation and added to the much longer main loop.

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char ** argv) {

    int i, j, k;
    int * A, * B, * Counts, ** Buckets, * GlobalCounts,
        * Offsets, * GlobalCountsTrans;
    int * C, * MyOffsets, * ArrayOffset, * Transmit, spread,
        target, * ReceiveSizes, * ReceiveOffsets;
    int max, globalmax;
    double logResult;
    int depth;
    int radix, multiple = 1;
    int rank, size;
    int n, d;
    FILE * number_file;
    char line[100];
    int * list;
    double starttime, endtime;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    n = (argc < 2) ? 1024 : atoi(argv[1]);
    d = (argc < 3) ? 8 : atoi(argv[2]);
    radix = size * d;

    list = (int *) malloc(sizeof(int) * n);
    A = (int *) malloc(sizeof(int) * n / size);
    B = (int *) malloc(sizeof(int) * n / size);
    GlobalCounts = (int *) malloc(sizeof(int) * radix * size);
    GlobalCountsTrans = (int *) malloc(sizeof(int) * radix * size);
    Counts = (int *) malloc(sizeof(int) * radix);
```

```

Buckets = (int **) malloc(sizeof(int *) * radix);
C = (int *) malloc(sizeof(int) * n / size);
Offsets = (int *) malloc(sizeof(int) * size * radix);
MyOffsets = (int *) malloc(sizeof(int) * radix);
ArrayOffset = (int *) malloc(sizeof(int) * size);
Transmit = (int *) malloc(sizeof(int) * size);
ReceiveSizes = (int *) malloc(sizeof(int) * size);
ReceiveOffsets = (int *) malloc(sizeof(int) * size);

/* read in the data */
if (rank == 0) {
    number_file = fopen("Random.txt", "r");

    if (number_file == NULL) {
        fprintf(stderr, "Could not read random number file.\n");
        MPI_Finalize();
        exit(1);
    }

    for (i = 0; i < n; i++) {
        fgets(line, sizeof(line), number_file);
        sscanf(line, "%d", &j);
        list[i] = j;
    }

    fclose(number_file);
}

starttime = MPI_Wtime();

MPI_Scatter(list, n / size, MPI_INT, A, n/size, MPI_INT, 0, MPI_COMM_WORLD);

/* they should be set to radix sort now... */
/* get the maximum */
max = 0;
for (i = 0; i < n / size; i++) {
    if (A[i] > max) max = A[i];
}

MPI_Allreduce(&max, &globalmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

logResult = log(globalmax) / log(radix);

depth = (logResult > (int) logResult) ? (int) logResult + 1 :

```



```

(int) logResult;

for (j = 0; j < depth; j++) {

    for (i = 0; i < radix; i++) {
        Counts[i] = 0;
    }

    for (i = 0; i < n / size; i++) {
        k = (A[i] / multiple) % radix;
        Counts[k]++;
    }

    Buckets[0] = &B[0];
    for (i = 1; i < radix; i++) {
        Buckets[i] = Buckets[i - 1] + Counts[i - 1];
        Counts[i - 1] = 0;
    }
    Counts[radix - 1] = 0;

    for (i = 0; i < n / size; i++) {
        k = (A[i] / multiple) % radix;
        Buckets[k][Counts[k]] = A[i];
        Counts[k]++;
    }

    /* build a global bucket lengths array with an all gather */
    MPI_Allgather(Counts, radix, MPI_INT, GlobalCounts, radix,
        MPI_INT, MPI_COMM_WORLD);

    /* transpose Global Counts array -- it is now row by processor,
        column by bucket, and needs to be column by processor, row by
        bucket */
    for (i = 0; i < size; i++) {
        /* i iterates on processor */
        for (k = 0; k < radix; k++) {
            /* k iterates on buckets */
            GlobalCountsTrans[k * size + i] = GlobalCounts[i * radix + k];
        }
    }

    /* do a global parallel prefix */
    Offsets[0] = 0;

```

```

for (i = 1; i < size * radix; i++) {
    Offsets[i] = Offsets[i-1] + GlobalCountsTrans[i-1];
}

/* pluck out this processor's offsets */
for (i = 0; i < radix; i++) {
    MyOffsets[i] = Offsets[i * size + rank];
}

/* now try to find where these things must go */
for (i = 0; i < size; i++) {
    Transmit[i] = 0;
    ArrayOffset[i] = 0;
}

for (i = 0; i < radix; i++) {
    target = MyOffsets[i] / (n / size);
    spread = MyOffsets[i] + Counts[i];

    if (spread > (target + 1) * (n / size)) {
        Transmit[target] += (target + 1) * (n / size) - MyOffsets[i];
        Transmit[target + 1] += spread - (target + 1) * (n / size);
    } else {
        Transmit[target] += Counts[i];
    }
}

for (i = 1; i < size; i++) {
    ArrayOffset[i] = ArrayOffset[i - 1] + Transmit[i-1];
}

/* spread around the amount you're going to send */
MPI_Alltoall(Transmit, 1, MPI_INT, ReceiveSizes, 1, MPI_INT, MPI_COMM_WORLD);

ReceiveOffsets[0] = 0;
for (i = 1; i < size; i++) {
    ReceiveOffsets[i] = ReceiveOffsets[i - 1] + ReceiveSizes[i - 1];
}

/* now do the real sending! */
MPI_Alltoallv(B, Transmit, ArrayOffset, MPI_INT,
    C, ReceiveSizes, ReceiveOffsets, MPI_INT, MPI_COMM_WORLD);

```

```

    for (i = 0; i < radix; i++) {
        Counts[i] = 0;
    }

    for (i = 0; i < n / size; i++) {
        k = (C[i] / multiple) % radix;
        Counts[k]++;
    }

    Buckets[0] = &A[0];
    for (i = 1; i < radix; i++) {
        Buckets[i] = Buckets[i - 1] + Counts[i - 1];
        Counts[i - 1] = 0;
    }
    Counts[radix - 1] = 0;

    for (i = 0; i < n / size; i++) {
        k = (C[i] / multiple) % radix;
        Buckets[k][Counts[k]] = C[i];
        Counts[k]++;
    }

    multiple *= radix;
}

MPI_Barrier(MPI_COMM_WORLD);
endtime = MPI_Wtime();

if (rank == 0) {
    printf("p %d n %d d %d time %f \n", size, n, d, endtime - starttime);
}

MPI_Finalize();
}

```

E.2 Minimum Spanning Tree Code

E.2.1 Structures

The following data structures are used in all the UPC implementations. The “edgeNo” field is used by the Boruvka algorithm as an index to a list of edges.

```
struct adjEntry {
    int vertex;
    int weight;
    int edgeNo;
};

struct edgeStruct {
    int vertex[2];
    int weight;
    int low_select;
    int high_select;
};
```

E.2.2 Prim-UPC-1

The following is the original implementation of Prim’s algorithm in UPC. We assume that a driver program loads the adjacency lists of the graph into AdjacencyList, ListCounts, and ListPtrs, and that there are n vertices in the graph.

```
void prim_mst(shared struct adjEntry * AdjacencyList,
    shared int * ListCounts,
    shared int * ListPtrs,
    int n,
    shared int * sum)
{
    int i, j, k;

    /* local copies and variables */
    struct adjEntry localMin;
    int localCount;
    int localSize;
    int localVerts[n];
    int localDistances[n];
```

```

int localListCounts[n];
int localListPtrs[n];
int localInTree[n];
struct adjEntry * localAdj;

/* shared arrays -- must be dynamically allocated */
/* because shared cannot go in activation record */
shared int * inTree;
shared struct adjEntry * minimums;
shared int * new;

/* allocate the shared structures needed for the algorithm */
inTree = (shared int *) upc_all_alloc(sizeof(shared int), n);
minimums = (shared struct adjEntry *)
upc_all_alloc(sizeof(shared struct adjEntry), THREADS);
new = (shared int *) upc_all_alloc(sizeof(shared int), 1);

/* use upc_forall to assign vertices to threads */
/* also get the count to size the localAdj */
k = 0;
localCount = 0;
localSize = 0;
upc_forall(i = 0; i < n; i++; &ListCounts[i]) {
    localVerts[localCount] = i;
    localCount++;
    localSize += ListCounts[i];
}

localAdj = (struct adjEntry *) malloc(sizeof(struct adjEntry) * localSize);

/* copy the adjacency list entries to local variables */
for (i = 0; i < localCount; i++) {
    /* build our own counts and pointers */
    localListCounts[i] = ListCounts[localVerts[i]];
    localListPtrs[i] = (i == 0) ? 0 : localListPtrs[i - 1] +
        localListCounts[i - 1];

    /* set the default distances */
    localDistances[i] = INT_MAX;

    /* copy the adjacency list entries */
    for (j = 0; j < localListCounts[i]; j++) {
        localAdj[j + localListPtrs[i]] =
            AdjacencyList[ListPtrs[localVerts[i]] + j];
    }
}

```

```

    }
}

/* kickstart Prim's by adding vertex 0 to the tree */
if (MYTHREAD == 0) {
    *new = 0;
    inTree[0] = 1;
    *sum = 0;
}

upc_barrier;

/* Prim's parallel algorithm */
for (i = 0; i < n - 1; i++) {

    /* find the minimum distance for my local vertices */
    localMin.weight = INT_MAX;
    localMin.vertex = -1;
    for (j = 0; j < localCount; j++) {

        /* update my local tree records if necessary */
        if (*new == localVerts[j]) {
            localInTree[j] = 1;
        }

        /* only consider un-added nodes */
        if (localInTree[j] != 1) {
            /* update for the latest new case */
            for (k = 0; k < localListCounts[j]; k++) {
                if (localAdj[k + localListPtrs[j]].vertex == *new) {
                    if (localDistances[j] > localAdj[k +
                        localListPtrs[j]].weight) {
                        localDistances[j] = localAdj[k +
                            localListPtrs[j]].weight;
                    }
                }
            }

            /* see if this is a new minimum */
            /* is <= so that we will suggest */
            /* a real vertex if we have one */
            if (localDistances[j] <= localMin.weight) {
                localMin.vertex = localVerts[j];
                localMin.weight = localDistances[j];
            }
        }
    }
}

```

```

        }
    }
}

/* post my new minimum to globally-accessible memory */
minimums[MYTHREAD] = localMin;

upc_barrier;

/* compute the actual minimum and post the newly-added vertex */
if (MYTHREAD == 0) {
    for (j = 1; j < THREADS; j++) {
        if (localMin.weight > minimums[j].weight || (
            localMin.weight == minimums[j].weight &&
            minimums[j].vertex >
            localMin.vertex)) {
            localMin = minimums[j];
        }
    }

    *new = localMin.vertex;
    if (localMin.weight < INT_MAX) {
        *sum += localMin.weight;
    }

}

upc_barrier;
}
}

```

E.2.3 Prim-UPC-2

In the second version of Prim's algorithm in UPC, we change the initial distribution of vertices so that it is balanced by number of edges, not number of vertices. The `upc_forall` loop is replaced by the following, and `start` and `end` now replace the `localVerts` array to indicate what vertices are owned by a particular thread.

```

/* copy the ListCounts */
sumEdgesOut = 0;
for (i = 1; i < n; i++) {

```

```

        localListCountsCopy[i] = ListCounts[i];
        sumEdgesOut += localListCountsCopy[i];
    }

    pivots[THREADS] = n;

    /* attempt to balance so everyone gets roughly sumEdgesOut/THREADS edges */
    for (j = 1, i = 0; i < THREADS; i++) {
        pivots[i] = j;
        k = 0;
        while (k < sumEdgesOut / THREADS && j < n) {
            k += localListCountsCopy[j];
            j++;
        }
        counts[i] = k;
    }

    start = pivots[MYTHREAD];
    end = pivots[MYTHREAD + 1];
    localCount = end - start;
    localSize = counts[MYTHREAD];

```

E.2.4 Prim-UPC-3

In the third version of Prim's algorithm in UPC, we add load balancing at regular intervals throughout the main loop. This oblivious load balancing is done by the following block of code, placed at the top of the main loop after the update to `localInTree`.

```

/* oblivious rebalance */
if (i % (n / THREADS) == 0 && i > 0 && i < n - 1 - THREADS) {

    /* preserve distances */
    for (j = 0; j < localCount; j++) {
        globalDistances[j + start] = localDistances[j];
    }

    /* get new sum of available edges */
    sumEdgesOut = 0;
    for (j = 1; j < n; j++) {
        if (!localInTree[j]) {
            sumEdgesOut += localListCountsCopy[j];
        }
    }
}

```



```

/* compute everyone's new piece of the array */
for (j = 1, r = 0; r < THREADS; r++) {
    pivots[r] = j;
    k = 0;
    while (k < sumEdgesOut / THREADS && j < n) {
        if (!localInTree[j]) { k += localListCountsCopy[j]; }
        j++;
    }
    counts[r] = k;
}

start = pivots[MYTHREAD];
end = pivots[MYTHREAD + 1];
localCount = end - start;
localSize = counts[MYTHREAD];

/* get and copy your piece of the adjacency list */
free(localAdj);
localAdj = (struct adjEntry *) malloc(sizeof(struct adjEntry) * localSize);

for (j = 0; j < localCount; j++) {
    localListCounts[j] = (localInTree[j + start] == 1) ? 0 :
        localListCountsCopy[j + start];
    localListPtrs[j] = (j == 0) ? 0 : localListPtrs[j - 1] +
        localListCounts[j - 1];

    if (!localInTree[j + start]) {
        localDistances[j] = globalDistances[j + start];
    }

    for (r = 0; r < localListCounts[j]; r++) {
        localAdj[r + localListPtrs[j]] = AdjacencyList[ListPtrs[j +
            start] + r];
    }
}
}

```

E.2.5 Prim-UPC-4

For this final implementation of Prim's algorithm in UPC, we change the condition of the load balancing step from one based on the iteration, and thus oblivious to the current work distribution, to one that monitors the work distribution and rebalances whenever a processor

becomes idle.

The following is added before the load balancing step in the main loop:

```
r = 0;
for (s = 0; s < THREADS; s++) {
    if (pivots[s] <= localNew && localNew < pivots[s + 1]) {
        r = s;
    }
}
counts[r]--;
```

The condition of the load balancing step is changed to:

```
if (counts[r] < 1 && i < n - n / THREADS)
```

The upper bound on i is provided to prevent thrashing as the algorithm completes.

E.2.6 Prim-MPI

The following is the MPI implementation of Prim's algorithm. This implementation scaled well for the test cases. It avoids explicit barrier synchronizations by using collective functions. From this we believe that UPC performance could be improved with the use of the recently-standardized collective operations once they are supported by UPC compilers. This code does have the somewhat unrealistic assumption that the adjacency list can be held in local memory on any particular processing element. However, unlike with UPC, no memory limit was encountered with MPI code on the T3e.

We assume that a driver program initializes and finalizes MPI, and that it loads the adjacency lists into ListCounts, ListPtrs, AdjListVerts, and AdjListWeights.

```
int * ListCounts;
int * ListPtrs;
int * AdjListVerts;
int * AdjListWeights;
int rank;
int threads;
int sum;
int n;
```

```

int mst() {

    int i, j, k;
    int * inTree;
    int edges;
    int start, end;
    int * scatterDispls;
    int * scatterSizes;
    int new;
    int * minimums;
    int localMins[2];
    int localCount;
    int * distances;
    int minimumWeight, minimumVertex;

    /* initialize sum */
    sum = 0;

    /* these are used for computing global information */
    scatterSizes = (int *) malloc(sizeof(int) * threads);
    scatterDispls = (int *) malloc(sizeof(int) * threads);
    minimums = (int *) malloc(sizeof(int) * 2 * threads);

    /* only 0 holds n at present */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* these are used for local copies of data */
    if (rank != 0) {
        ListCounts = (int *) malloc(sizeof(int) * n);
        ListPtrs = (int *) malloc(sizeof(int) * n);
        inTree = (int *) malloc(sizeof(int) * n);
        distances = (int *) malloc(sizeof(int) * n);
    }

    /* we assume these starting values -- nothing in tree, */
    /* everything at maximal distance */
    for (i = 0; i < n; i++) {
        inTree[i] = 0;
        distances[i] = INT_MAX;
    }

    /* spread around the adjacency list details */
    MPI_Bcast(ListCounts, n, MPI_INT, 0, MPI_COMM_WORLD);

```

```

MPI_Bcast(ListPtrs, n, MPI_INT, 0, MPI_COMM_WORLD);

/* prepare to receive the adjacency list from 0 */
if (rank != 0) {
    k = 0;
    for (i = 0; i < n; i++) {
        k += ListCounts[i];
    }
    edges = k;

    AdjListVerts = (int *) malloc(sizeof(int) * k);
    AdjListWeights = (int *) malloc(sizeof(int) * k);
}

for (i = 0; i < threads; i++) {
    scatterSizes[i] = rank * n / threads;
}
scatterSizes[threads - 1] += n % threads;

start = scatterSizes[rank];
end = (rank == threads - 1) ? n : scatterSizes[rank + 1];
localCount = end - start;

/* receive the adjacency list from 0 */
MPI_Bcast(AdjListVerts, edges, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(AdjListWeights, edges, MPI_INT, 0, MPI_COMM_WORLD);

/* kickstart Prim's algorithm by adding 0 to the tree */
new = 0;

/* Parallel Prim's */
for (i = 0; i < n - 1; i++) {

    /* Tell everyone the latest added vertex */
    MPI_Bcast(&new, 1, MPI_INT, 0, MPI_COMM_WORLD);

    inTree[new] = 1;

    /* find the new vertex at a minimal distance from the tree */
    localMins[0] = -1;
    localMins[1] = INT_MAX;

    for (j = start; j < end; j++) {

```

```

        if (inTree[j] != 1) {
            for (k = ListPtrs[j]; k < ListPtrs[j] + ListCounts[j]; k++) {
                if (AdjListVerts[k] == new) {
                    if (AdjListWeights[k] <= distances[j]) {
                        distances[j] = AdjListWeights[k];
                    }
                }
            }

            if (distances[j] < localMins[1]) {
                localMins[0] = j;
                fflush(stdout);
                localMins[1] = distances[j];
            }
        }

    }

    /* everyone tell 0 what your new minimum is */
    MPI_Gather(&localMins[0], 2, MPI_INT, minimums,
              2, MPI_INT, 0, MPI_COMM_WORLD);

    /* 0 computes the new vertex to add to the tree */
    if (rank == 0) {
        minimumVertex = minimums[0];
        minimumWeight = minimums[1];
        for (j = 1; j < threads; j++) {
            if (minimums[2 * j + 1] < minimumWeight) {
                minimumVertex = minimums[2 * j];
                minimumWeight = minimums[2 * j + 1];
            }
        }

        new = minimumVertex;
        if (minimumWeight < INT_MAX) {
            sum += minimumWeight;
        }
    }
}
}
}

```

E.2.7 Boruvka-UPC

The following is the first implementation of Boruvka's algorithm in UPC. The algorithm performed poorly, and does not attempt to take advantage of locality or minimize communication.

```
void mst_2(shared struct adjEntry * AdjacencyList,
          shared int * ListCounts,
          shared int * ListPtrs,
          int n, shared int * sum)
{
    shared struct edgeStruct * edgeList;
    int i, j, k, r, s, t, offsets[THREADS];
    int iter;
    int count, offset;
    int edges;
    int components;
    struct adjEntry tempAdj;
    shared int * lowCount;
    shared int * ppArray;
    struct edgeStruct tempEdge;
    shared int * edgeChoice;
    shared int * componentVertexMap;
    shared int * componentVertexCount;
    shared int * componentVertexPtrs;
    shared int * newComponentMap;
    shared int * newAdjacencyListCounts;
    shared int * newAdjacencyListPtrs;
    shared struct adjEntry * newAdjacencyList;
    shared int * localSums;

    /* allocate the various shared arrays needed */
    ppArray = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);
    localSums = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);
    lowCount = (shared int *) upc_all_alloc(sizeof(shared int), n);
    edgeChoice = (shared int *) upc_all_alloc(sizeof(shared int), n);
    componentVertexMap = (shared int *) upc_all_alloc(sizeof(shared int), n);

    /* we repeat until our tree is too small! */
    while (n > 1) {

        upc_forall(i = 0; i < THREADS; i++; &ppArray[i]) {
            ppArray[i] = 0;
            localSums[i] = 0;
```

```

}

upc_forall(i = 0; i < n; i++; &lowCount[i]) {
    lowCount[i] = 0;
    edgeChoice[i] = 0;
    componentVertexMap[i] = 0;
}

upc_barrier;

/* PHASE 1: Make an edge list */

/* get a count of how many edges we have. Further, lowCount will
   be the offset to the edges in the edge list such that those
   edges for which i is low will start at lowCount[i] */

localSums[MYTHREAD] = 0;

    offset = MYTHREAD * ( n / THREADS );
count = n / THREADS;

if (MYTHREAD == THREADS - 1) {
    count += n % THREADS;
}

ppArray[MYTHREAD] = 0;
for (i = offset; i < offset + count; i++) {
    lowCount[i] = 0;
    for (j = ListPtrs[i]; j < ListCounts[i] + ListPtrs[i]; j++) {
        if (AdjacencyList[j].vertex > i) {
            lowCount[i]++;
        }
    }
    ppArray[MYTHREAD] += lowCount[i];
    if (i > offset) {
        lowCount[i] += lowCount[i - 1];
    }
}

upc_barrier;

edges = 0;
offsets[0] = 0;
for (i = 0; i < THREADS; i++) {

```

```

        edges += ppArray[i];
        if (i != THREADS - 1) {
            offsets[i+1] = offsets[i] + ppArray[i];
        }
    }

    for (i = offset; i < offset + count; i++) {
        lowCount[i] += offsets[MYTHREAD];
    }

    edgeList = (shared struct edgeStruct *)
        upc_all_alloc(sizeof(shared struct edgeStruct), edges);

    /* link low */
    k = offsets[MYTHREAD];
    tempEdge.low_select = 0;
    tempEdge.high_select = 0;
    for (i = offset; i < offset + count; i++) {
        for (j = ListPtrs[i]; j < ListCounts[i] + ListPtrs[i]; j++) {
            if (AdjacencyList[j].vertex > i) {
                tempEdge.vertex[0] = i;
                tempEdge.vertex[1] = AdjacencyList[j].vertex;
                tempEdge.weight = AdjacencyList[j].weight;
                AdjacencyList[j].edgeNo = k;
                edgeList[k] = tempEdge;
                k++;
            }
        }
    }

    upc_barrier;

    /* link high */
    for (i = offset; i < offset + count; i++) {
        for (j = ListPtrs[i]; j < ListCounts[i] + ListPtrs[i]; j++) {
            if (AdjacencyList[j].vertex < i) {
                if (AdjacencyList[j].vertex == 0) {
                    s = 0;
                } else {
                    s = lowCount[AdjacencyList[j].vertex - 1];
                }
                r = lowCount[AdjacencyList[j].vertex];

                for (k = s; k < r; k++) {

```



```

        if (edgeList[k].vertex[1] == i) {
            AdjacencyList[j].edgeNo = k;
        }
    }
}

upc_barrier;

/* PHASE 2: Everyone, grab your lightest edge! */

for (i = offset; i < count + offset; i++) {
    r = -1;
    s = INT_MAX;
    edgeChoice[i] = -1;
    for (j = ListPtrs[i]; j < ListCounts[i] + ListPtrs[i]; j++) {

        if (AdjacencyList[j].weight < s) {
            r = AdjacencyList[j].vertex;
            s = AdjacencyList[j].weight;
            edgeChoice[i] = AdjacencyList[j].edgeNo;
        }

    }

    if (r != -1) {
        localSums[MYTHREAD] += s;
        if (r < i) {
            edgeList[edgeChoice[i]].high_select = 1;
        } else {
            edgeList[edgeChoice[i]].low_select = 1;
        }
    }
}

upc_barrier;

/* add our partial sums to the running total */
if (MYTHREAD == 0) {
    for (i = 0; i < THREADS; i++) {
        *sum += localSums[i];
    }
}

```

```

}

/* forward everyone to the root of their tree */
upc_forall(i = 0; i < n; i++; &edgeChoice[i]) {
    while(edgeList[edgeChoice[i]].high_select !=
          edgeList[edgeChoice[i]].low_select) {

        if (edgeList[edgeChoice[i]].high_select) {
            edgeChoice[i] = edgeChoice[edgeList[edgeChoice[i]].vertex[0]];
        } else {
            edgeChoice[i] = edgeChoice[edgeList[edgeChoice[i]].vertex[1]];
        }
    }
}

upc_barrier;

/* count the number of components that you hold */
/* also, fix the nagging overcount on the local sums */
localSums[MYTHREAD] = 0;
ppArray[MYTHREAD] = 0;
for (i = offset; i < offset + count; i++) {
    if(edgeChoice[i] != -1 && edgeList[edgeChoice[i]].vertex[0] == i &&
        edgeList[edgeChoice[i]].low_select == 1 &&
        edgeList[edgeChoice[i]].high_select == 1) {

        ppArray[MYTHREAD]++;
        localSums[MYTHREAD] += edgeList[edgeChoice[i]].weight;
    }
}

upc_barrier;

/* add our partial sums to the running total */
if (MYTHREAD == 0) {
    for (i = 0; i < THREADS; i++) {
        *sum -= localSums[i];
    }
}

```

```

components = 0;
for (i = 0; i < THREADS; i++) {
    components += ppArray[i];
}

if (components == 0) break;

/* PHASE 3: Identify the new components, and map vertexes to components */

newComponentMap = (shared int *) upc_all_alloc(sizeof(shared int),
    components);
newAdjacencyListCounts = (shared int *) upc_all_alloc(sizeof(shared int),
    components);
newAdjacencyListPtrs = (shared int *) upc_all_alloc(sizeof(shared int),
    components);

componentVertexCount = (shared int *) upc_all_alloc(sizeof(shared int),
    components);
componentVertexPtrs = (shared int *) upc_all_alloc(sizeof(shared int),
    components);

if (MYTHREAD == 0) {
    for (i = 1; i < THREADS; i++) {
        ppArray[i] += ppArray[i - 1];
    }
}

upc_barrier;

k = (MYTHREAD == 0) ? 0 : ppArray[MYTHREAD - 1];
for (i = offset; i < count + offset; i++) {
    if(edgeList[edgeChoice[i]].vertex[0] == i) {
        newComponentMap[k] = edgeChoice[i];
        k++;
    }
}

upc_barrier;

upc_forall(i = 0; i < components; i++; &newComponentMap[i]) {
    componentVertexCount[i] = 0;

    for (j = 0; j < n; j++) {
        if (edgeChoice[j] == newComponentMap[i]) {

```

```

        componentVertexCount[i]++;
    }
}

upc_barrier;

if (MYTHREAD == 0) {
    componentVertexPtrs[0] = 0;
    for (i = 1; i < components; i++) {
        componentVertexPtrs[i] = componentVertexPtrs[i - 1] +
            componentVertexCount[i - 1];
    }
}

upc_barrier;

upc_forall(i = 0; i < components; i++; &newComponentMap[i]) {

    componentVertexCount[i] = 0;

    for (j = 0; j < n; j++) {
        if (edgeChoice[j] == newComponentMap[i]) {
            componentVertexMap[componentVertexPtrs[i] +
                componentVertexCount[i]] = j;
            componentVertexCount[i]++;
        }
    }
}

upc_barrier;

/* PHASE 4: Build the new adjacency matrix */
/* now, at last, we build a new adjacency matrix */

upc_forall(i = 0; i < components; i++; &newAdjacencyListCounts[i]) {

    newAdjacencyListCounts[i] = 0;

    for (j = 0; j < components; j++) {
        t = 0;
        if (j != i) {

```

```

        for (r = componentVertexPtrs[j]; r < componentVertexPtrs[j] +
            componentVertexCount[j] && t == 0; r++) {

            for (s = ListPtrs[componentVertexMap[r]];
                s < ListPtrs[componentVertexMap[r]] +
                    ListCounts[componentVertexMap[r]] && t == 0; s++) {

                for (k = componentVertexPtrs[i];
                    k < componentVertexPtrs[i] +
                        componentVertexCount[i] &&
                            t == 0; k++) {

                    if (AdjacencyList[s].vertex ==
                        componentVertexMap[k]) {
                        t = 1;
                    }
                }
            }
        }
        newAdjacencyListCounts[i] += t;
    }
}

upc_barrier;

if (MYTHREAD == 0) {
    newAdjacencyListPtrs[0] = 0;
    for (i = 1; i < ppArray[THREADS - 1]; i++) {
        newAdjacencyListPtrs[i] = newAdjacencyListPtrs[i - 1] +
            newAdjacencyListCounts[i - 1];
    }
}
upc_barrier;

newAdjacencyList = (shared struct adjEntry *) upc_all_alloc(
    sizeof(shared struct adjEntry),
    newAdjacencyListPtrs[components - 1] +
    newAdjacencyListCounts[components - 1]);

/* build the list */
/* now, at last, we build a new adjacency matrix */
upc_forall(i = 0; i < components; i++; &newAdjacencyListCounts[i]) {

```

```

newAdjacencyListCounts[i] = 0;

for (j = 0; j < components; j++) {
    tempAdj.vertex = j;
    tempAdj.weight = INT_MAX;
    t = 0;
    if (j != i) {

        for (r = componentVertexPtrs[j]; r < componentVertexPtrs[j] +
            componentVertexCount[j]; r++) {

            for (s = ListPtrs[componentVertexMap[r]];
                s < ListPtrs[componentVertexMap[r]] +
                ListCounts[componentVertexMap[r]]; s++) {

                for (k = componentVertexPtrs[i];
                    k < componentVertexPtrs[i] +
                    componentVertexCount[i]; k++) {

                    if (AdjacencyList[s].vertex ==
                        componentVertexMap[k]) {
                        t = 1;
                        if (AdjacencyList[s].weight < tempAdj.weight) {
                            tempAdj.weight = AdjacencyList[s].weight;
                        }
                    }
                }
            }
        }
    }
    if (t > 0) {
        newAdjacencyList[newAdjacencyListPtrs[i] +
            newAdjacencyListCounts[i]] = tempAdj;
        newAdjacencyListCounts[i]++;
    }
}

n = components;

upc_barrier;

upc_free(AdjacencyList);

```

```

        AdjacencyList = newAdjacencyList;
        upc_free(ListCounts);
        ListCounts = newAdjacencyListCounts;
        upc_free(ListPtrs);
        ListPtrs = newAdjacencyListPtrs;

    }

    upc_free(componentVertexMap);
    upc_free(lowCount);
    upc_free(edgeChoice);
    upc_free(ppArray);
    upc_free(localSums);
}

```

E.3 Low Level Code Tests

E.3.1 Read Time

The following was used to measure the time to read from shared memory in UPC on the Cray T3e.

```

shared int bigArray[8 * 1024 * 1024];

int main() {

    int i, j, k, m, qt;
    shared int * mediumArray;
    int littleArray[1024 * 1024];
    clock_t start[20];
    struct tms timeStruct;

    for(i = 1024; i < 8 * 1024 * 1024; i += 2) {

        start[3] = 0;
        for (qt = 0; qt < 10; qt++) {
            if (MYTHREAD == 0) start[0] = times(&timeStruct);
            for (k = 0, j = MYTHREAD * i / THREADS; k < i; k++, j++) {

```

```

        j = j % i;
        m = j % (1024 * 1024);
        littleArray[m] = bigArray[j];
    }
    upc_barrier;

    if (MYTHREAD == 0) {
        start[1] = times(&timeStruct);
        start[3] += start[1] - start[0];
    }
    upc_barrier;

}
upc_barrier;

if (MYTHREAD == 0) {
    printf("%d %d \n", i, (int)(start[3]/10));
}
upc_barrier;
}
upc_barrier;
}

```

E.3.2 Write Time

The following was used to measure the time to write to shared memory in UPC on the Cray T3e.

```

shared int bigArray[8 * 1024 * 1024];

int main() {

    int i, j, k, m, qt;
    shared int * mediumArray;
    int littleArray[1024 * 1024];
    clock_t start[20];
    struct tms timeStruct;

    for(i = 1024; i < 8 * 1024 * 1024; i += 2) {

        start[3] = 0;
        for (qt = 0; qt < 10; qt++) {

```



```

        if (MYTHREAD == 0) start[0] = times(&timeStruct);
        for (k = 0, j = MYTHREAD * i / THREADS; k < i; k++, j++) {
            j = j % i;
            m = j % (1024 * 1024);
bigArray[j] = j;
        }
        upc_barrier;

        if (MYTHREAD == 0) {
            start[1] = times(&timeStruct);
            start[3] += start[1] - start[0];
        }
        upc_barrier;

    }
    upc_barrier;

    if (MYTHREAD == 0) {
        printf("%d %d \n", i, (int)(start[3]/10));
    }
    upc_barrier;
}
upc_barrier;
}

```

E.3.3 MPI Latency Test

The following is a quick MPI latency test. A single integer is sent between two threads. The time printed is that for sending the message and receiving the reply.

```

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <sys/times.h>
#include <time.h>

int main(int argc, char * argv[]) {

    int rank, temp;
    MPI_Status status;
    double starttime, endtime;

```

```

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
    starttime = MPI_Wtime();
    MPI_Send(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&temp, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    endtime = MPI_Wtime();
    printf("Round-trip time: %f %f \n", endtime - starttime,
        CLK_TCK * (endtime - starttime));
    printf("Clock tick %d\n", CLK_TCK);
} else {
    MPI_Recv(&temp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

MPI_Finalize();
}

```

E.3.4 MPI Communication Cost Tests

The following was used to measure the cost for larger payloads in MPI.

```

#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <sys/times.h>
#include <time.h>
#include <sys/types.h>

int main(int argc, char * argv[]) {

    int bigArray[8 * 1024 * 1024];
    int bigArray2[4 * 1024 * 1024];
    int temp;
    double sum, processor_sum;
    int i, j, k;
    double starttime, endtime;
    int rank, count;
    MPI_Status status;

```

```

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &count);

if (rank == 0) {
    printf("Sequential send times: \n");
}

for (i = 1024; i < 16 * 1024 * 1024; i = i + i) {
    if (rank == 0) {
        sum = 0;
        for (j = 1; j < count; j++) {
            processor_sum = 0;
            for (k = 0; k < 10; k++) {
                starttime = MPI_Wtime();
                MPI_Send(&bigArray, i, MPI_INT, j, 0, MPI_COMM_WORLD);
                MPI_Recv(&temp, 1, MPI_INT, j, 0, MPI_COMM_WORLD, &status);
                endtime = MPI_Wtime();
                processor_sum += endtime - starttime;
            }
            printf("P0 to P%d Size %d Time %f \n",
                j, i, processor_sum / 10.0 * CLK_TCK);
            sum += processor_sum;
        }
        printf("P0 average Size %d Time %f \n",
            i, sum / 10.0 / (count - 1) * CLK_TCK);
    } else {
        for (k = 0; k < 10; k++) {
            MPI_Recv(&bigArray, i, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Send(&temp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }
}

MPI_Finalize();
}

```

This was modified to measure time for collective operations:

```

#include <stdio.h>
#include <mpi.h>

```

```

#include <stdlib.h>
#include <sys/times.h>
#include <time.h>
#include <sys/types.h>

int main(int argc, char * argv[]) {

    int * bigArray, * bigArray2;
    int temp;
    double sum, processor_sum;
    int i, j, k;
    double starttime, endtime;
    int rank, count;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &count);

    bigArray = (int *) malloc(sizeof(int) * 4 * 1024 * 1024);
    bigArray2 = (int *) malloc(sizeof(int) * 4 * 1024 * 1024);

    for (i = 1024; i < 8 * 1024 * 1024; i = i + i) {
        sum = 0;
        for (k = 0; k < 10; k++) {
            if (rank == 0) {
                starttime = MPI_Wtime();

                MPI_Alltoall(bigArray, i / count, MPI_INT, bigArray2, i / count, MPI_INT,
                    MPI_COMM_WORLD);

                if (rank == 0) {
                    endtime = MPI_Wtime();
                    sum += endtime - starttime;
                }
            }

        }

        if (rank == 0) {
            printf("All to all for array size %d time %f \n", i, sum / 10.0 * CLK_TCK);
            fflush(stdout);
        }
    }
}

```

```

    MPI_Finalize();

}

```

E.4 Common Function Implementations

E.4.1 Bucketing

```

void Bucketing(int * localSrc, int * localDest, int * localBuckets[],
    int length, int * Count, int range, int (*getkey)(int)) {
    int i, j, k;
    int offsets[range]; /* the offset in localDest corresponding to
                        each bucket in localBuckets */
    /* zero our counters */
    for (i = 0; i < range; i++) {
        Count[i] = offsets[i] = 0;
    }
    /* find out the quantity that go in each bucket */
    for (i = 0; i < length; i++) {
        Count[(getkey)(localSrc[i])]++;
    }
    /* build offsets as a parallel prefix of counts */
    /* while zeroing the counters */
    offsets[0] = 0;
    for (j = 1; j < range; j++) {
        offsets[j] = offsets[j - 1] + Count[j - 1];
        Count[j - 1] = 0;
    }
    Count[range - 1] = 0;
    /* construct the buckets as pointers into localDest using
       our computed offsets */
    localBuckets[0] = &localDest[0];
    for (j = 1; j < range; j++) {
        localBuckets[j] = localBuckets[0] + offsets[j];
    }
    /* at last, put everything from localSrc into a bucket */
    for (i = 0; i < length; i++) {
        k = (getkey)(localSrc[i]);
        *(localBuckets[k] + Count[k]++) = localSrc[i];
    }
}

```

E.4.2 Thread Prefix

Two variants were produced – one in which the final array was allocated by the function call, and one in which this was passed to the function as an argument.

```
shared int * ThreadPrefix(int S) {

    int i;
    // shared int * segment_size;
    shared int * target_indices;
    // segment_size = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);
    TPH_segment_size[MYTHREAD] = S;

    target_indices = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);
    target_indices[MYTHREAD] = 0;
    for (i = 0; i <= MYTHREAD; i++) {
        target_indices[MYTHREAD] += TPH_segment_size[i];
    }

    upc_barrier;

    return target_indices;
}

void ThreadPrefix(shared int * target_indices, int S) {

    int i;
    shared int * segment_size;

    segment_size = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);
    segment_size[MYTHREAD] = S;

    for (i = 0; i <= MYTHREAD; i++) {
        target_indices[MYTHREAD] += segment_size[i];
    }

    upc_barrier;
    if (MYTHREAD == 0) upc_free(segment_size);
}
```

E.4.3 Generalized Thread Prefix

The thread prefix code above always performed an addition. This generalization was developed to use an arbitrary binary function of two integers.

```
shared int * ThreadPrefixExt(int S, int (*func)(int, int)) {

    int i;
    shared int * target_indices;
    shared int * TPEH_segment_size;

    TPEH_segment_size = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);

    TPEH_segment_size[MYTHREAD] = S;

    target_indices = (shared int *) upc_all_alloc(sizeof(shared int), THREADS);
    target_indices[MYTHREAD] = 0;
    for (i = 0; i <= MYTHREAD; i++) {
        target_indices[MYTHREAD] = (func)(target_indices[MYTHREAD],
            TPEH_segment_size[i]);
    }

    upc_barrier;

    return target_indices;
}
```

E.4.4 Thread Concatenate

```
void ThreadConcatenate(shared int * target, int * Source, int S) {
    int i, start;
    shared int * target_indices;

    target_indices = ThreadPrefix(S);
    start = (MYTHREAD == 0) ? 0 : target_indices[MYTHREAD - 1];

    for (i = 0; i < S; i++) {
        target[start + i] = Source[i];
    }

    upc_barrier;
}
```