Computer Science Technical Report

Implementing Sort in UPC: Performance and Optimization

Kohinoor (Lisa) Begum and Steven Seidel

Michigan Technological University
Computer Science Technical Report
CS-TR-05-06
2005

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

Abstract

Unified Parallel C (UPC) is a parallel extension of ANSI C that is based on a partitioned shared address space. It supports development of parallel applications over diverse hardware platforms and does not require a true shared memory system. It simplifies programming by providing the same syntax for local references as for remote references. This helps the user focus on performance, and allows for the exploitation of locality. Our project is to implement a generic sort function in UPC. It will help programmers focus on design, implementation and performance of the entire application rather than work on details of sorting.

`upc_all_sort` is an implementation of a generic sort function in UPC using a bin sort algorithm. Techniques applied to improve the performance of the sort function include: sampling of data to ensure more nearly uniform bucket sizes among threads, an all-to-all exchange of bucket sizes to minimize the amount of dynamically allocated memory, rebalancing data at the end of the sort to ensure that each thread finishes with the same number of elements as it started with. In this paper, we analyze and compare the performance of sequential and parallel sort, the potential of performance improvements over sequential sort and also the bottlenecks limiting these improvements. This sort function runs 3.9 times faster than the sequential sort for problem sizes of at least $128K$ when run on 8 threads. The performance scales up with the increasing number of threads and bigger problem sizes. It runs 6.8 times faster than sequential sort for problem size of $512K$ when run on 16 threads.

Contents

List of Figures

1. Introduction

Unified Parallel C (UPC) is a parallel extension of ANSI C that is based on a partitioned shared address space [1]. It supports development of parallel applications over diverse hardware platforms and does not require a true shared memory system. Sorting is an extensively studied problem in computer science and has numerous practical applications. Providing an implementation of a generic sort function in UPC will help programmers focus on design, implementation and performance of the entire application rather than work on details of sorting.

2. UPC sort function definition

`upc_all_sort` was first described in the UPC collective operations specifications v1.0 [2], but it was deprecated at the most recent UPC workshop. It is not practical to implement a generic sort function that delivers similar performance over all possible source data distributions. But it is still helpful to have an implementation that has reasonable speedup in most common cases.

The function prototype of `upc_all_sort` is as follows:

```
#include <upc.h>
#include <upc_collective.h>

upc_all_sort(shared void *A, size_t elem_size, size_t nelems,
     size_t blk_size,int (func)(shared void *, shared void *),
     upc_flag_t sync_mode);
```

This function sorts `nelems` elements of `elem_size` bytes starting at `A`. A `blk_size` of zero represents the indefinite array layout and a nonzero `blk_size` represents that the array has a blocking factor of `blk_size`. The function `func` is the user-defined comparison function. It returns an integer less than, equal to, or greater than zero if first argument is considered to be respectively less than, equal to, or greater than the second. `sync_mode` specifies synchronization requirements at initiation and completion of the sort. `upc_all_sort` performs an in place sort and at completion `A` contains the sorted elements. `upc_all_sort` does not perform a *stable* sort. The reason behind this is given in section 3.4.

The `upc_all_sort` function exploits locality by using *affinity*. All threads view shared memory as logically partitioned into `THREADS` blocks. M_i is the block of shared memory that has

affinity to thread i. Any element in M_i is local to thread i, whereas any element in M_j is remote to thread i, where $i \neq j$. Remote references often take much longer than local references. Each thread has a private address space that is only accessable by itself. Any element in private memory of thread i is only accessable to thread i. More details about UPC can be found in the HP UPC Programmers' Guide [4].

In `upc_all_sort`, threads start working with their local shared elements. The layout of source data among threads complicates the implementation and it affects the performance of the sort. How the array is distributed among threads depends on the `blocksize`. It is hard to achieve better performance for all possible array layouts. Several data layout possibilities are presented in Figure 1.

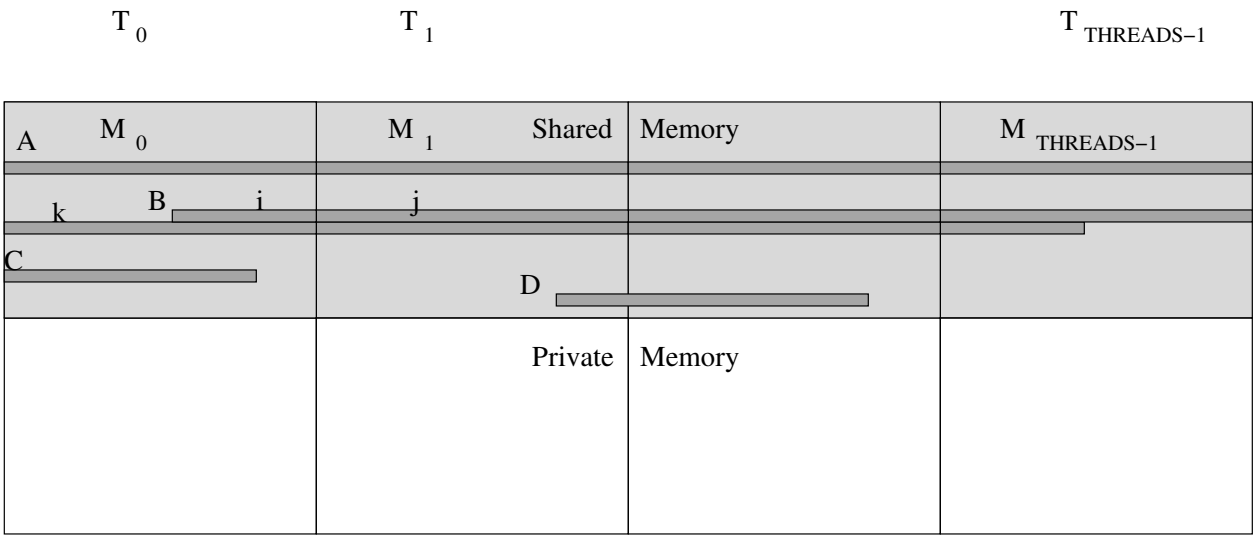

Figure 1. Different Array distributions

In Figure 1, `A` is a uniformly distributed array that has blocksize $[*]$. `upc_all_sort` will perform well with `A` since all threads have an equal number of elements in their shared local memory to work with, and the load is balanced among all threads. `B` is an array with more than `THREADS` blocks, so it wraps around. `upc_all_sort` will perform well with `B` but it has to respect the phase and wrapping of blocks to make sure it does not try to copy or access something out of bounds. `C` is allocated entirely in the local shared space of only thread 0. `D` is a case where only two threads have all the data. Such cases where all the data reside in one or a couple of threads might perform poorly. Initially, threads that the source elements have affinity to are active and others sit idle, which hampers speedup. We may be able to improve performance by redistributing the source data before calling `upc_all_sort`. This motivates the need for a wrapper for redistribution. We have added a new flag to the function prototype, where the user can choose to redistribute the source data.

The new function prototype is:

```
#include <upc.h>
#include <upc_collective.h>

upc_all_sort(shared void *A, size_t elem_size, size_t nelems,
             size_t blk_size,int (func)(shared void *, shared void *),
             upc_flag_t sync_mode, upc_flag_t rdst);
```

If `rdst` is 1, our implementation redistributes the data among threads into a dynamically allocated array with block size [*]. The new array `R` is equavalent to the following declaration:

```
shared [ceil(nelem/THREADS) * elem_size] char R[nelem*elem_size];
```

When the sort is complete, the data are redistributed back to the original source array. Even though this adds some overhead at the beginning and end it might improve performance by balancing the load among all threads. The user knows how the source data is laid out and it is his/her decision whether redistribution will help.

Implementing `upc_all_sort` as a generic sort function has several challenges, these include:

- *Source Pointer:* The source array to be sorted is of type `shared void *shared`. The data type of the source array is unknown. This disallows the use of assignment operators for copying data. Incrementing `*src` by 1 does not move to the next element, but rather to the next byte. Pointer arithmetic is required to find out the correct offsets each time pointing to an element to make sure the pointer is not pointing to some unrealistic address.
- *User defined comparison function:* `upc_all_sort` uses a user defined comparison function for comparing elements. This function is called each time a comparison is done.
- *Blocksize:* Blocksize determines the data layout of the source array. The generic sort function has to consider all different possible block sizes and make sure that it works correctly for each case. Especially, it has to be careful about block sizes when threads are copying data and ensure that they do not try to copy elements bigger than the block size at a time.

3. Design of the sort implementation

The generic sort function is implemented using a parallel bin sort algorithm [5]. The following are the basic steps in `upc_all_sort`:

- Local Sort
- Bucketing
- Exchange
- Merge

3.1 Local Sort

Each thread uses a `shared [] char *shared` to access its local shared elements. Therefore, it views its local shared data as a single consecutive block even if the source data are wrapped around. Each thread sorts its local shared elements using a quicksort. Before going to the bin sort, threads check whether all the source data have affinity to a single thread. If all data belong to a single thread then sorting is complete and threads should return without further execution.

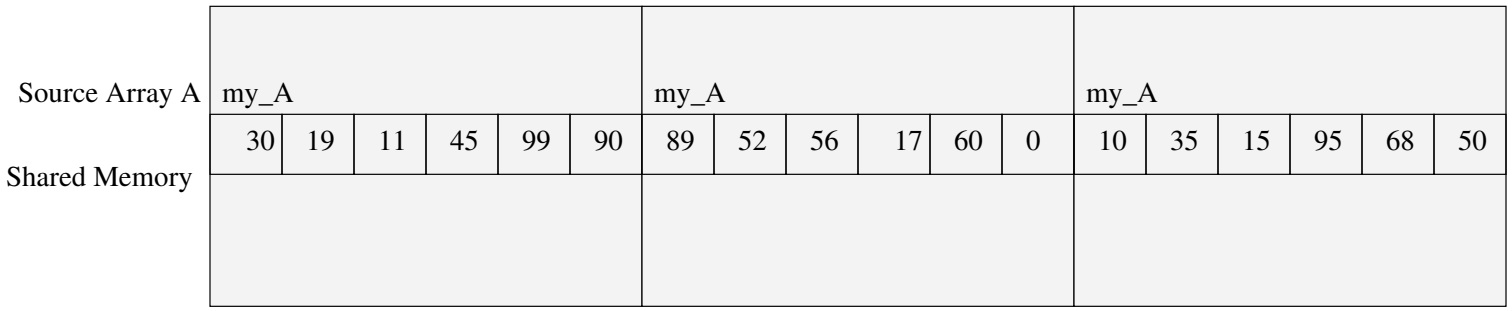

Figure 2. Source Array

Figure 2 shows the source array `A` is uniformly distributed among all threads. Figure 3 shows the source array after local sort.

3.2 Bucketing

Each thread works with its local shared elements and divides them into buckets or bins. $THREADS-1$ number of elements are chosen for partitioning the range of source elements. These elements

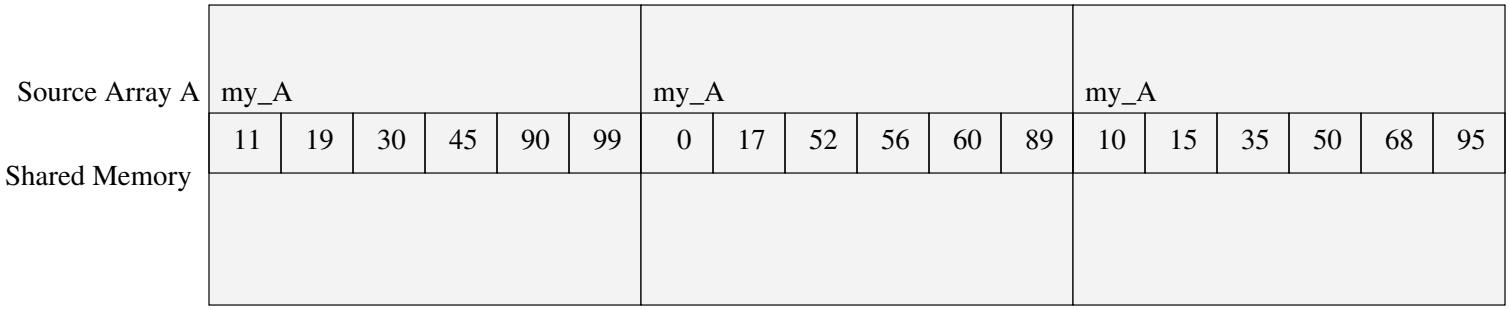

Figure 3. Local Sort

are called splitting keys. Thread 0 chooses these splitting keys and broadcasts it to all other threads using an array `PARTITION` in shared memory. Detail information about how splitting keys are chosen is provided in section 4.1. Each thread has a copy of the splitting keys in its local shared space. Comparing with the splitting keys, each thread maps the elements into `THREADS` bins. All elements in bin i are less than all elements in bin $i+1$ for `THREADS` $> i \geq 0$. There are `THREADS`^{2} bins and each bin is denoted as b_{ij} , where thread i is the consumer of the bin and thread j is the producer of the bin. Notice that the elements are not put into any physical bins in this stage. Only the starting indices of each bin and the count of elements are stored in a shared array. Threads use this array in the next step to actually exchange bins. For instance, thread j has n elements. It logically maps the elements into `THREADS` different bins and stores the associated bin count in `C`. Therefore, `C[i][j]`contains the number of elements thread i maps for its bucket j.

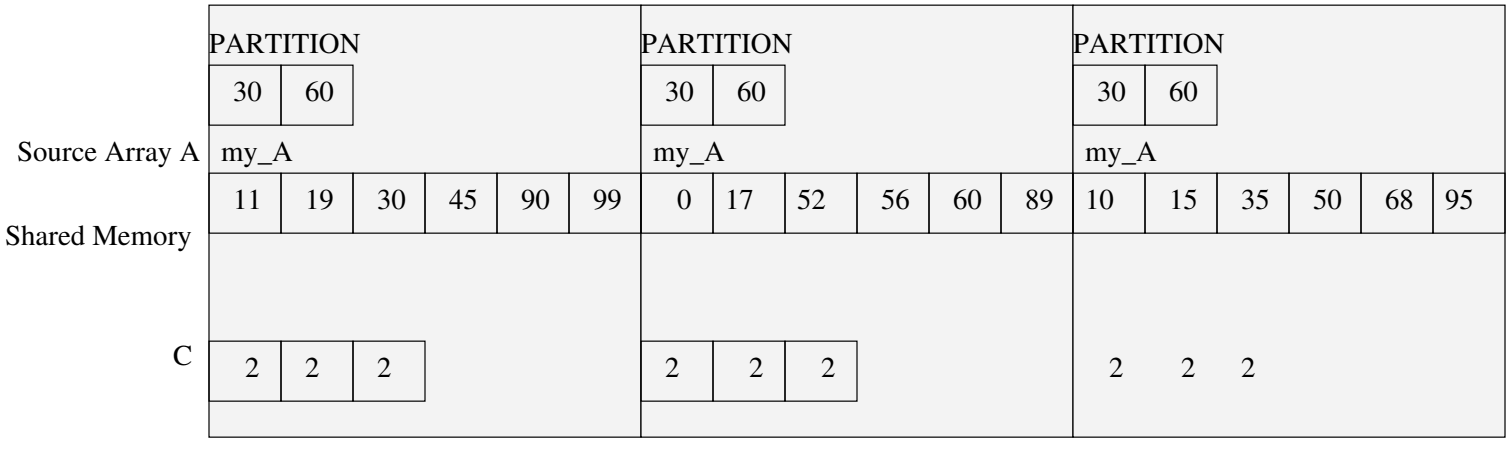

Figure 4. Bucketing

In Figure 4, `PARTITION` contains the splitting keys. Each thread i works on its local shared

elements and using the splitting keys, it stores the count of elements of each bucket j in `C[i][j]`.

3.3 Exchange

All threads exchange buckets. From `C`, all threads know how many elements they will get from each other. Each thread computes this total number of elements by performing a logical transpose, and prefix on the counts stored in `C`. After these computations, each thread i has the total number of elements in `C[i][THREADS-1]` and creates an array `B` of exactly that size to hold all the bins it will recieve from other threads. In all circumstances, the number of elements in `B` is twice the number of elements in `A`. The extra space in B is needed for use as temporary place holders in merging bins. Each thread i performs a right shift on the ith row of `C`. Now, `C[i][j]` contains the offset thread i has to place the bin it pulls from thread j. Then threads pull buckets (using `upc_memcpy`) from other threads preserving the bucket ordering. Thread i pulls `C[i][j+1] - C[i][j]` elements from thread j and places it after b_{ij-1} in shared array `B`. Therefore, in array `B`, thread i has $b_{i0}, b_{i1}, \ldots, b_{ij}, \ldots, b_{i\texttt{THREADS}-1}$. Threads have to respect the phase so that they do not try to access something out of bounds. If thread i sees `C[i][j+1] - C[i][j]` has zero count, it does not communicate with thread j to pull any elements.

Exchanging of bins among threads can be done by calling `upc_all_exchange` function. The `upc_all_exchange` function provided in the UPC collectives library requires all threads to call the function with the same arguments values. In cases, where the bin sizes are different it creates a problem. It requires threads to call the exchange function to exchange bins of size equal to the largest number of elements a thread receives from all others, but this might waste much space. We have implemented a version of the exchange function that is able to exchange variable length of blocks of data among threads. Therefore, each thread copies the exactly number of elements and it does not copy anything if the bucket size is zero.

In Figure 5, Each thread i knows where to place b_{ij} copied from thread j. Each thread i copies b_{ij} from thread j.

3.4 Merge

After the exchange, each thread has `THREADS` bins. Each bin contains the sorted elements. All elements in b_{ix} is smaller than elements in b_{jx} for every $i < j$. It merges these sorted subsequences preserving ascending order. Therefore, `B` contains `nelems` elements sorted. The last step is to rebalance and copy the sorted data back to the source array. Each thread locally computes where to place its first element and how many elements to place as consecutive

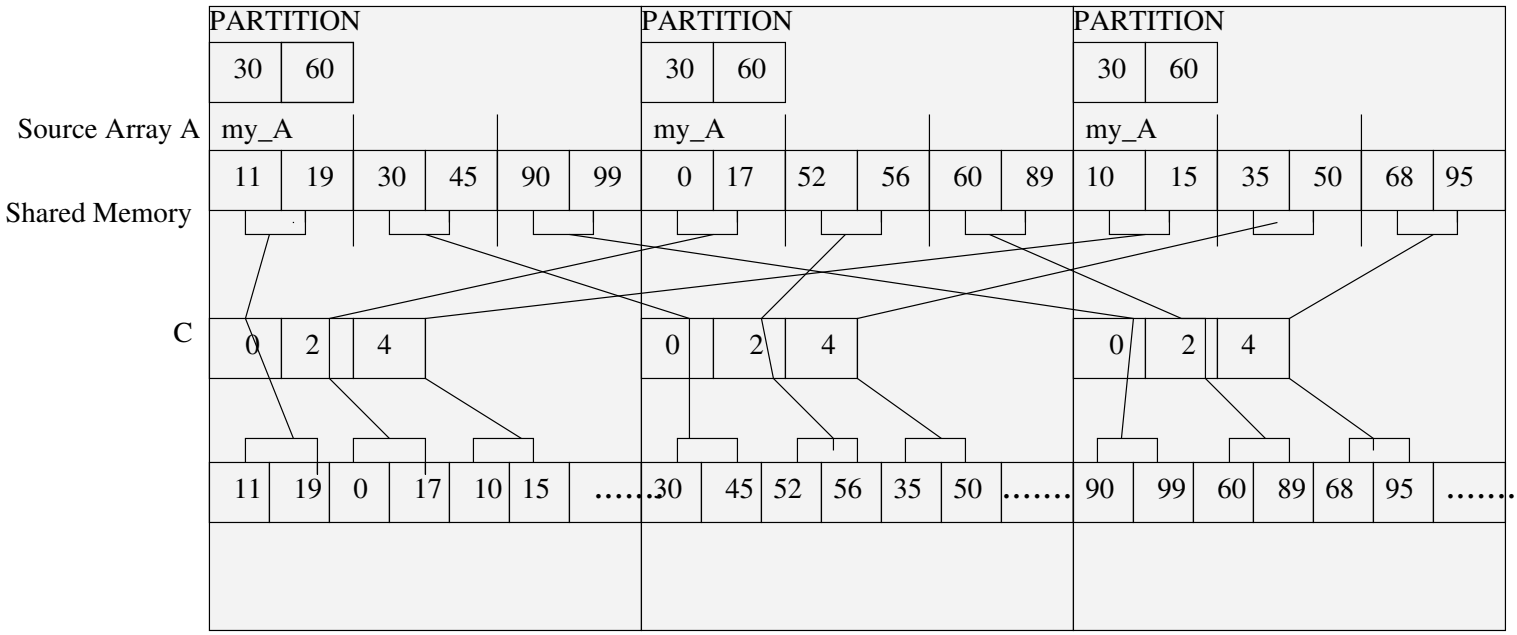

Figure 5. Exchange

items in `A`. Then it copies those many elements into `A` using `upc_memcpy`. It uses phase and blocksize of `A` to compute this element count to make sure in `upc_memcpy`, it does not try to copy across thread boundaries. Using the same process, it copies all its elements back into the source array `A`. The source array contains the sorted elements and each thread finishes with the same number of elements as it started with.

`upc_all_sort` does not perform a *stable* sort. It may reorder elements which are equal. For instance, in Figure 1, `B` has the same element at positions i,j, and k and the `upc_all_sort` function is called to sort `B`. In the bucketing step, these elements should be in bin for thread i. Thread 1 recieves those elements in exchange step. Thread 1 pulls elements at positions i,k from thread 0 places it in b_{10} and places element at j position in b_{11}. After merge and rebalance, all elements are sorted and ith element appears before kth element, and kth element appears before jth element. Even though the elements are equal, they are reordered since the elements reside in different threads.

3.5 Non-local Reference and Synchronization

All threads need to synchronize at different steps of the sort function. Synchronization is crucial when threads access elements in remote shared refernces. All threads need to make remote reference in time of choosing splitting keys, in the exchange step and again at the end

when rebalancing the data. All threads are synchronized these times by placing a barrier.

4. Improvement and Optimization

4.1 Load Balancing

Load balancing is a key factor in achieving good performance from parallel applications. If one thread does all the work and others sit idle, the program does not show performance improvements. Special attention has been given to balance the load among threads. In the `upc_all_sort` function, work is divided among threads by bucketing. Threads start working with local shared elements, partition elements into bins, exchange bins, and then work only on bins recieved. To ensure uniform load, we have to choose splitting keys very carefully. A naïve approach is to look at the elements to sort, find the *minimum* and *maximum* and then compute the splitting keys. This approach works fine if the data to be sorted are uniformly distributed over the *min - max* range but it may show poor result when the data are not uniformly distributed.

A sample-sorting technique has the potential for performance benefits and scalability in UPC [3]. We have applied sampling to ensure that all threads get a similar amount of work in most cases. Each thread quicksorts its local shared elements and sends a sample of `THREADS` elements to thread 0 from its local shared portion. These elements are chosen so that they evenly partition the local shared elements. This is implemented using `upc_all_gather`. Thread 0 sorts these samples and then chooses `THREADS`-1 evenly spaced splitting keys from it. Then thread 0 broadcasts these global splitting keys to all threads so that each thread has a copy. This sampling provides better load balance in cases of unknown or skewed distribution.

4.2 Binary Search

Each thread is responsible for mapping its elements into bins. A naïve approach is to go through the element one by one, compare each with the splitting keys and then decide which bucket it belongs to. Each thread has n elements, and `THREADS`-1 splitting keys. So this approach takes in worst-case $n($`THREADS`$-1)$ time units. In most cases, n is much larger than `THREADS`, so the cost of bucketing using the naïve approach is in $O(n)$.

We have implemented a binary search to reduce the cost of bucketing. We take advantage of the fact that all the data local to a single thread is sorted before bucketing. Instead of going through the elements one by one, we search for the splitting keys one by one and decide the bucket boundaries. Once we find the nearest value of the splitting key s_i we know all elements before s_i belong to b_i and anything later including this element belong to b_j, where j is greater

than i. Searching for splitting keys takes $(\texttt{THREADS} - 1)O(\log n)$ time units, so the cost is $O(\log n)$ under the assumption that $\log n >$ THREADS.

4.3 Exchange

In this step each thread is exchanging bucket with all other threads. Here, each thread has a bucket for itself. Instead of using upc_memcpy for copying bucket from thread i's source data to its bin b_{ii}, the pointer to bins are casted to be local and memcpy is used. Since memcpy is regular C function it takes less time than upc_memcpy.

4.4 Merge

B is the array that holds all the concatenated buckets from all other threads preserving bucket order. b_{ij} is the bucket thread i receives from thread j. In B, elements of b_{ij} always come before b_{ik} for every $j < k$. The contents of each bucket are already sorted when it is received by a thread. We take into account the fact these are sorted subsequences and merge them keeping the elements in ascending order. It takes $O(n \log \texttt{THREADS})$ time.

4.5 Local references

In UPC, threads can access any part of the global shared address space, but remote references often take longer than local references. When some element in remote shared memory is accessed often, it increases the run time. To avoid these run time delays local references are used whenever possible. If a thread needs to access some element in remote shared memory, it copies that element into its own shared address space. Thus it avoids remote references most of the time. For example, once thread 0 finds the splitting keys, it broadcasts the keys to all threads so that each thread has a copy of the splitting keys in their own shared address space. Therefore, in the binary search threads do not make any remote reference. Throughout the merge, most references are local.

5. Performance

The upc_all_sort function has been tested on on a Linux Myrinet cluster using MuPC compiler and on the HP SC40 Alpha Server using HP's UPC compiler with various block sizes, problem sizes, numbers of threads, and with and without redistribution. The HP server has eight four-processor nodes and uses a Quadric interconnect. Two versions of upc_all_sort are tested: using func, the user-defined comparison function, and using hard coded integer

comparison operator. Performance is measured by the run time of the sort. Measurements plotted in following graphs are averages of the maximum runtime of all threads on the HP server over 50 runs.[1]

Performance of `upc_all_sort` is described in the following two divisions:

5.1 Problem Sizes

`upc_all_sort` is tested with various problem sizes. For each problem size, the runtime of `upc_all_sort` is compared with that of a sequential sort. Thread 0 performs a quicksort on `N` local shared elements. That measurement is used as the baseline cost of a sequential sort. The runtime of `upc_all_sort` using the function `func` is compared with that of sequential sort using the same comparison function. Similarly, the runtime of `upc_all_sort` using a hard coded comparison is compared with sequential sort using hard coded comparison. Thus we have produced two sets of graphs for different problem sizes. In both sets, sequential sort and `upc_all_sort` is given the src pointer as `shared void *` and inside the sort this shared pointer is casted to a pointer to local. It improves the runtime in both sequential and parallel sort significantly. In `upc_all_sort`, the steps of local sort and merge is performed by using assignment operators in place of memcpy. But it does not improve performance significantly. These tests were run on `N` elements with block size `[*]`. Redistribution of data does not improve performance in cases of `blk_size [*]`. Therefore, the measurements presented in these graphs report the runtime without redistribution.

Sequential sort with a hard coded comparison operator takes less time than sort with the user function `func`, since making a call to a function for each comparison is time consuming. The runtime of `upc_all_sort` includes the time spent on sorting and a fixed overhead. This overhead comes from local computation, and communication and synchronization among threads. This overhead is fixed over any type of comparison method used. Therefore, sort with hard coded comparison achieves better performance than sort with user-defined function. This pattern can be followed throughout the figures in this section.

The `upc_all_sort` function is tested with problem sizes of $32K$ and larger. The runtime of sequential sort is similar to the runtime of the `upc_all_sort` function when run on 1 thread. Figure 6 shows the average of maximum runtimes of all threads sorting $32K$ elements with the function `func`. It shows performance improvement by a factor of 3 over sequential sort, when running on at least 8 threads. For larger problem size, the performance scales up, since the fixed overheard hides in the time we gain by dividing large number of elements. The runtime

[1]Runtime we obtained on Lionel is higher than the runtime on HP since MuPC on lionel adds a layer of translating upc function calls to MPI function calls where as on HP no layer is added. To show speed up we present data from HP here.

of our sort function is 4 times faster than base runtime with problem size $512K$ when run on 8 threads. Also, the performance scales up as the number of threads increases. As seen in Figure 6, `upc_all_sort` has a factor of 6.8 performance improvement over sequential with problem size $512K$ when run on 16 threads. Figure 7 shows result from the same test runs

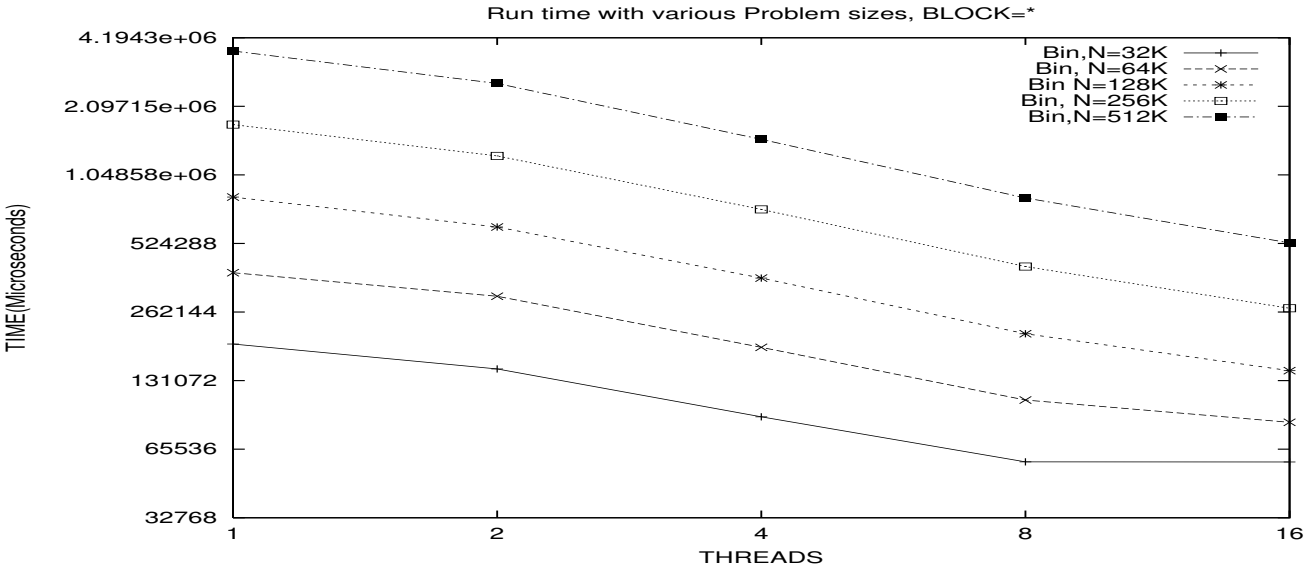

Figure 6. Average Runtime with different problem sizes (`func`)

plotting the minimum of maximum run times of all threads. The time variation between these two figures is not noticable. [2]

The sequential sort with hard coded comparison operator runs faster and, as mentioned above, the performance improvement of `upc_all_sort` is smaller in this case. As in Figure 8, `upc_all_sort` has the same execution time as sequential sort when run on 16 threads for $512K$. Figure 9, shows the minimum of maximum runtimes among threads. Figure 7 shows that `upc_all_sort` does not scale up to show any speed up for any probem size. This speed up is limited by the overhead in the `upc_all_sort` function.

`upc_all_sort` (with the function `func`) improves performance by a factor of 3 over sequential sort with large problem sizes. Figure 10 shows the speed up for various problem sizes. All scale up to a speed up between 3 to 6.8 for bigger problem sizes when run on 8 or 16 threads.

[2]Since the time variation between minimum and average of the maximum runtimes are not significant, it can be said that the maximum of the maximum runtimes are also close to average of the maximum runtimes. Therefore, the maximum of these runtimes are not presented in a seperate graph here.

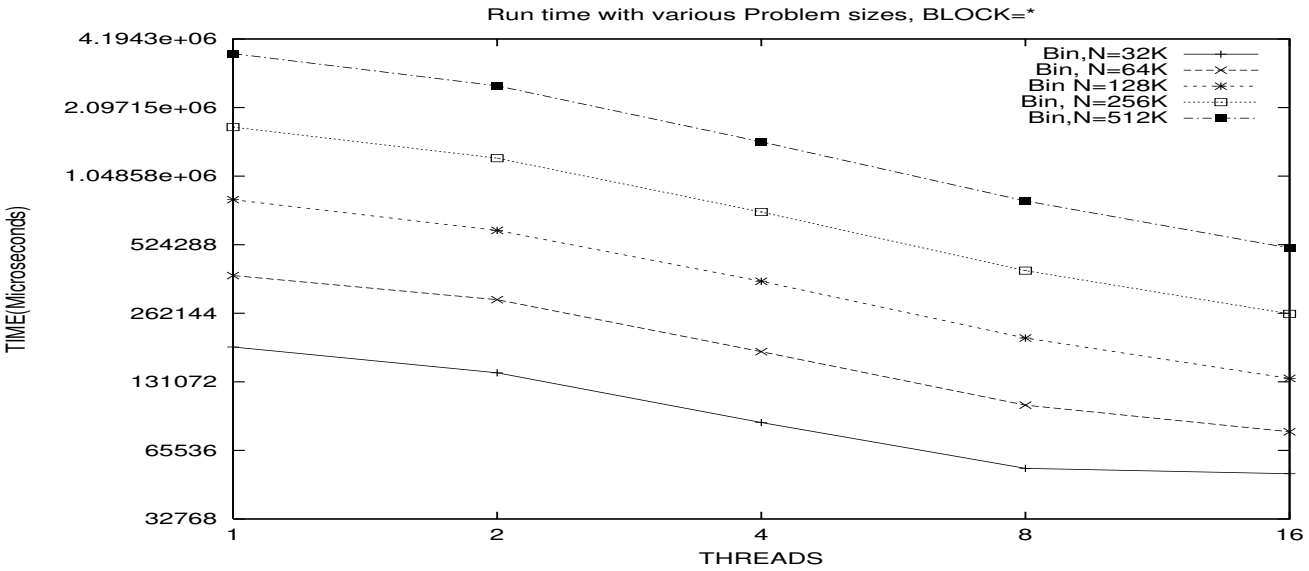

Figure 7. Minimum Runtime with different problem sizes (`func`)

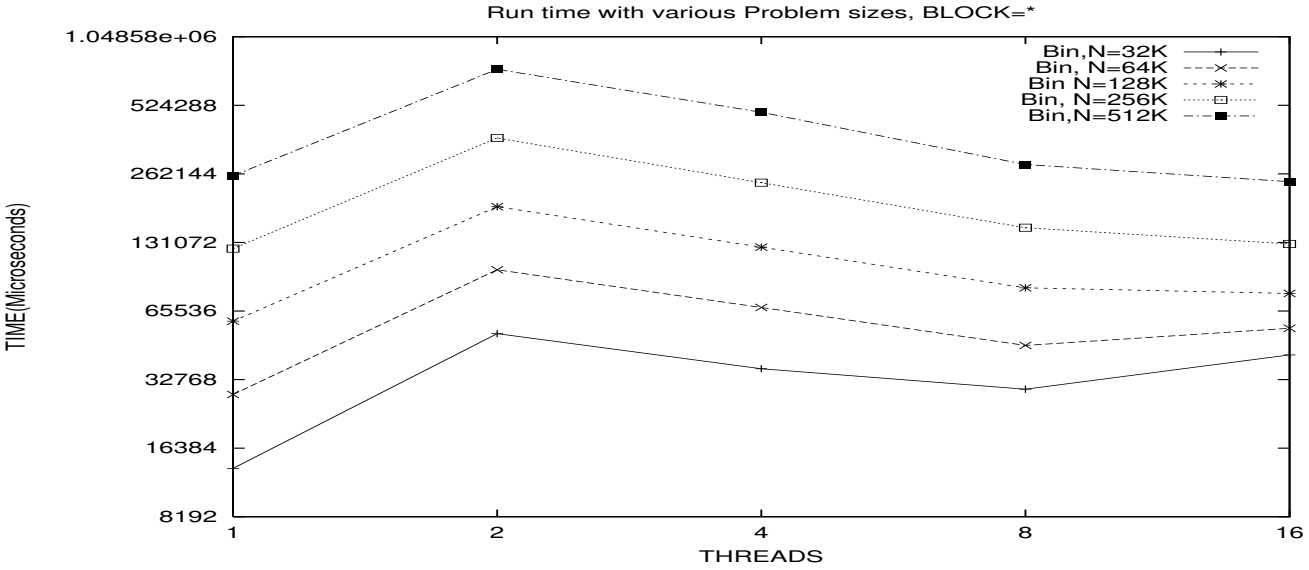

Figure 8. Average Runtime with different problem sizes (hard coded comparison operator)

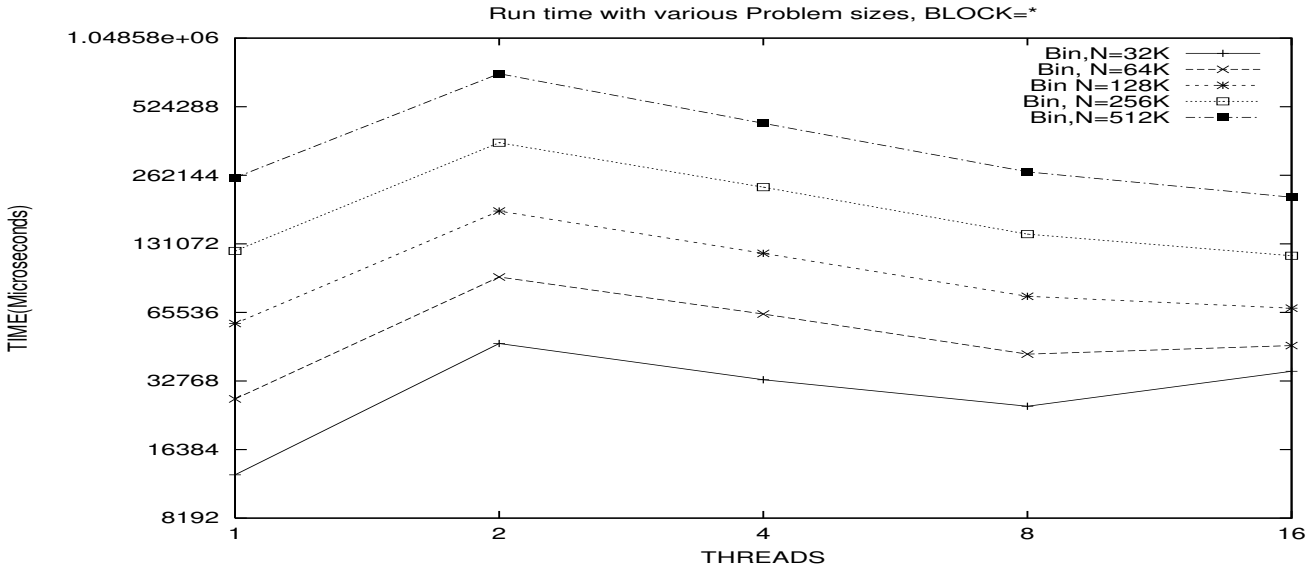

Figure 9. Minimum Runtime with different problem sizes (hard coded comparison operator)

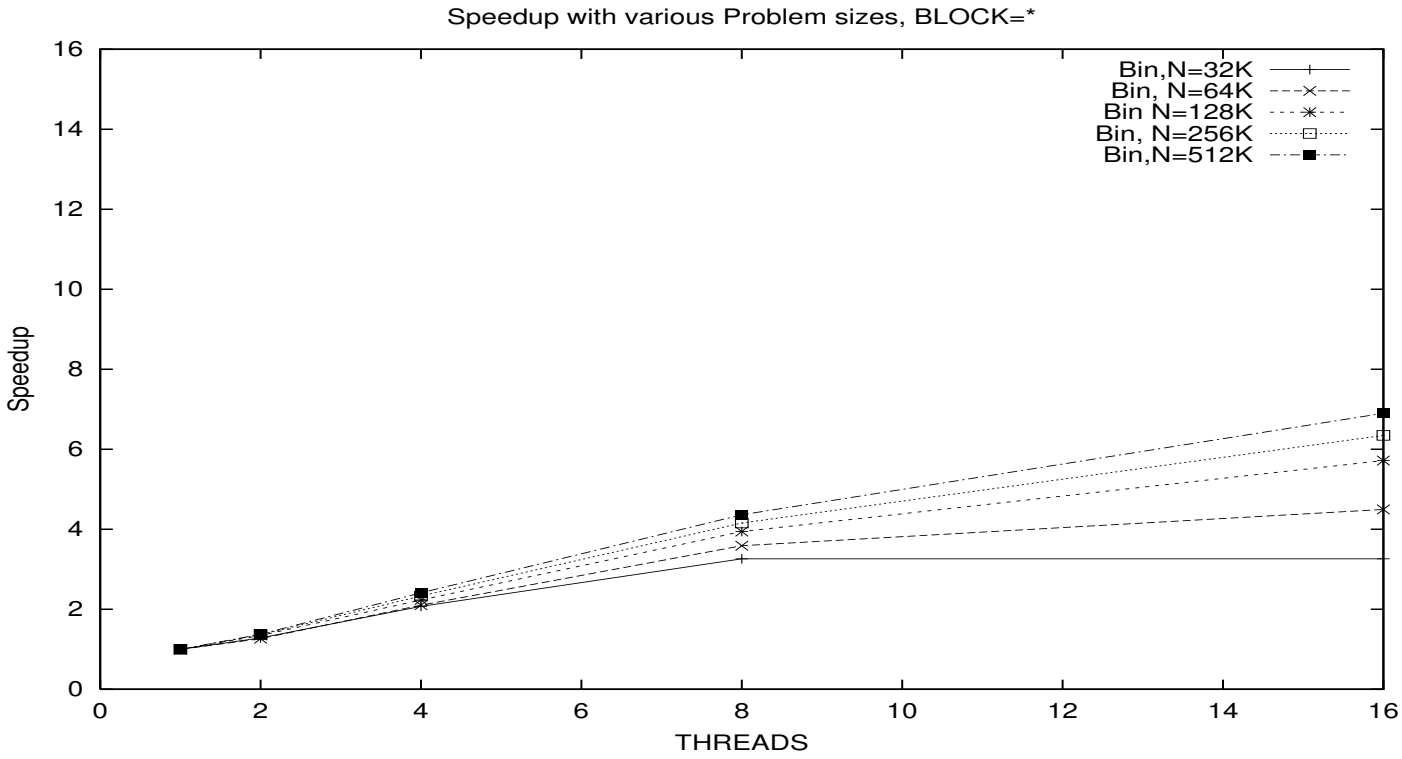

Figure 10. Runtime with different N

For further improvement, future work may include modifying in rebalance step. Currently in the rebalance each thread i may use more than one `upc_memcpy` to place back data to thread j if the data to be sent to j are not consecutive elements on thread i. This may happen when the source data wraps around threads. For example, this occurs when the source array is like B as described in section 2 (see Figure 1). Shuffling the elements at thread i so that all elements destined for thread j are contiguous and using a single `upc_memcpy` may improve the performance.

5.2 Block Sizes

The layout of the source data among threads affects the performance of `upc_all_sort`. If one or several threads calling the sort function have zero elements in its local shared portion, the runtime increases. In such cases, redistributing the data may improve the performance. We have tested `upc_all_sort` with various blocksizes and observed when it helps to redistribute the source data. The graphs presented here are runtimes with the problem size fixed at $128K$ and various blocksizes. They show runtimes both with and without redistribution. These are performance measurements from `upc_all_sort` using the function `func`. Graphs showing the performance measurements from test runs with `upc_all_sort` using hard coded comparison operator are given in the appendix A.

Figure 11 shows the runtimes for blocksize `[*]`. The sort wrapper redistributes source data into a new array that has block size `[*]`. In this case the redistribution does not improve the runtime. Furthermore, it adds a slight overhead on top of the regular sorting time. Therefore, redistributing a large array that has blocksize `[*]` is not recommended.

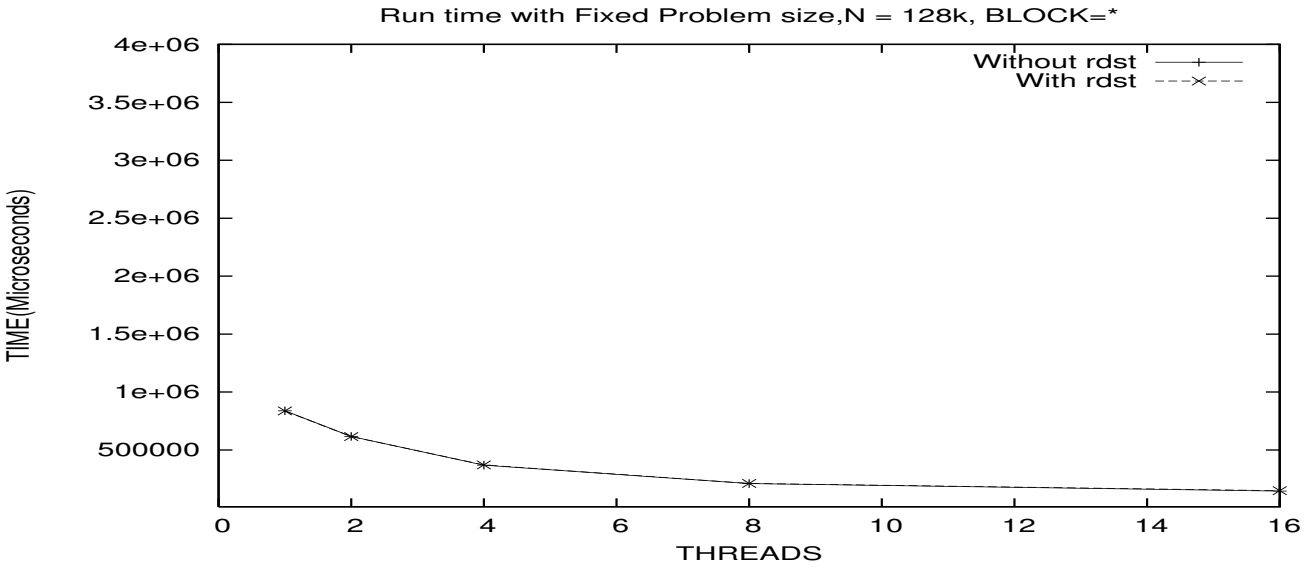

Figure 11. Runtime with BLOCK [*] (`func`)

In the array with blocksize 1, all threads have the same number of elements (plus or minus one) than others but the local shared elements are not contiguous. Redistribution of the data does not change the number of local shared elements in each thread. It places the local shared data in a contiguous block and also has to distribute sorted elements back to the source array after

sorting is complete. Here, the overhead of redistributing data and distributing it back is too high to get any performance benefit over sorting without redistribution. The overhead in distributing the data into the source array is higher than for blocksize [*]. This is why in Figure 12, the function with redistribution has higher runtime than function without redistribution, whereas no difference is noticed in Figure 11.

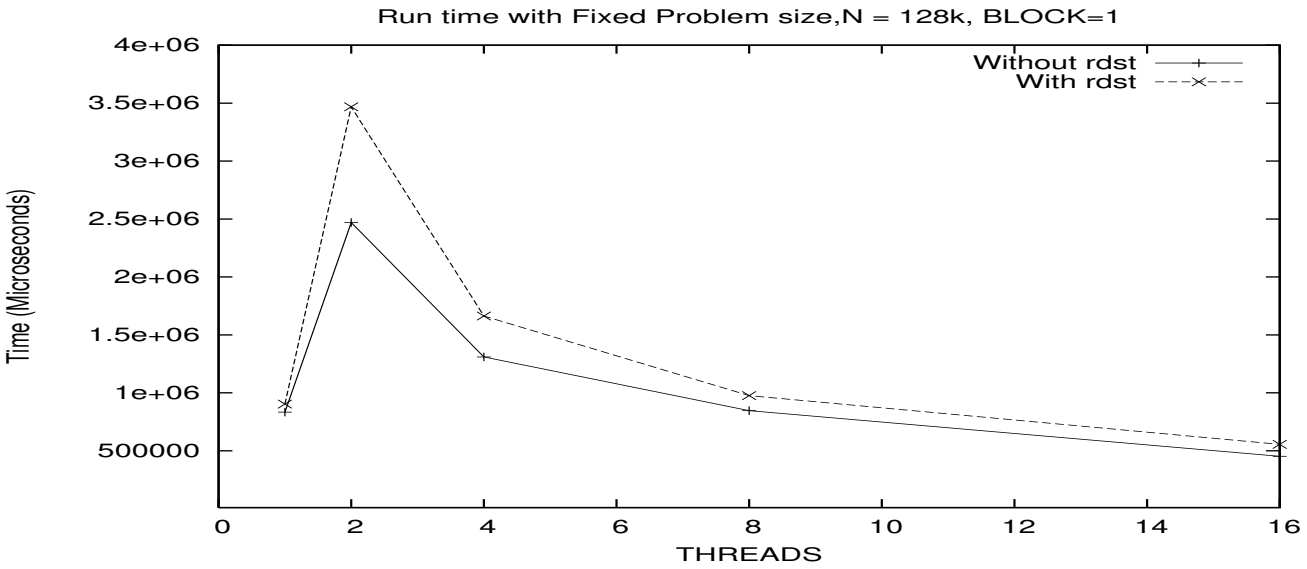

Figure 12. Runtime with BLOCK 1 (`func`)

When all source elements have affinity to a single thread, `upc_all_sort` behaves poorly if the redistribution option is not chosen. In this case only a single thread performs the sequential sort, but it has to synchronize with all other threads before returning from the sort function. This makes the `upc_all_sort` runtimes slower than the sequential run time. Figure 13 shows the run time with block size `N`, that is, all the data is in a single block on thread 0.

The runtime of sorting without redistribution does not decrease as the number of threads increases because only a single thread is working and others sit idle. Sorting with redistribution reduces the runtimes dramatically as the number of threads increases. The same behavior is observed in Figure 14, when all the data has affinity to a small number of threads. Redistribution helps even when only one thread has no elements and the other threads each have a large number of elements. For problem sizes of at least $64K$ redistribution of the source data improves performance over sorting without redistribution of data. In Figure 15, this performance difference between with and without redistribution is not as big as seen in Figure 13, where the blocksize is `N`.

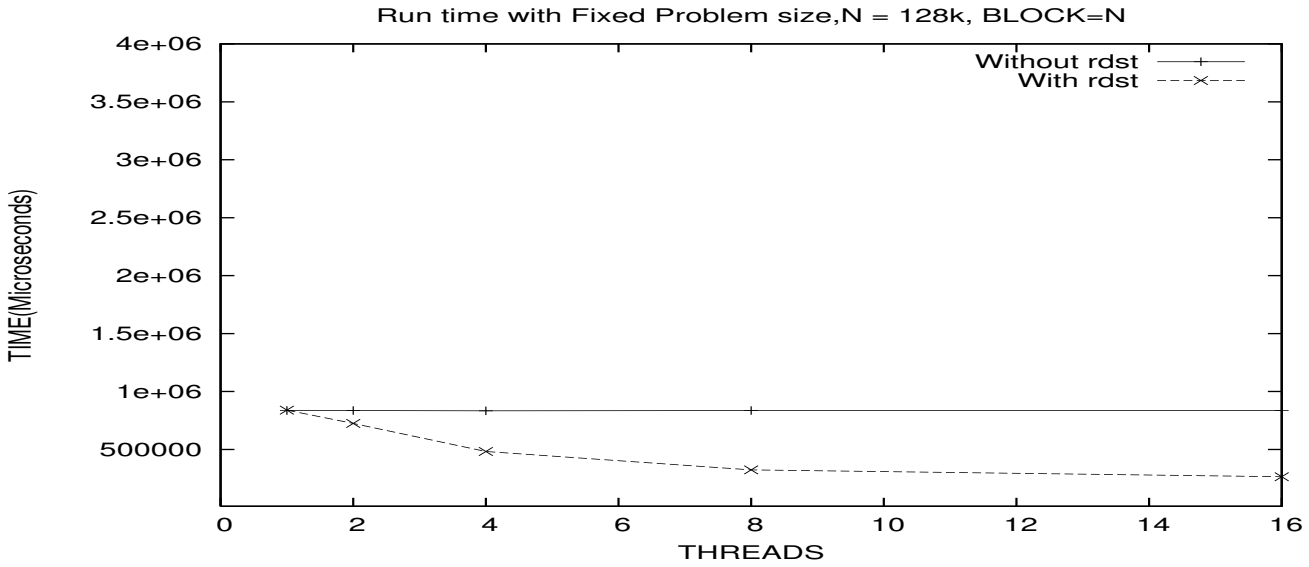

Figure 13. Runtime with BLOCK N (`func`)

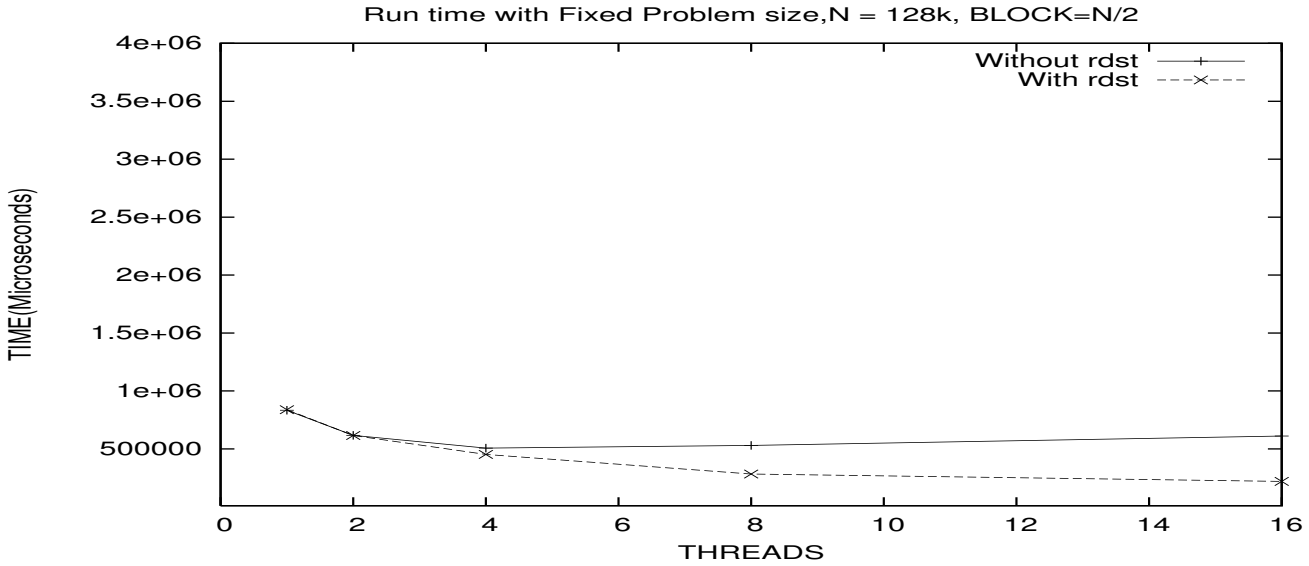

Figure 14. Runtime with BLOCK N/2 (`func`)

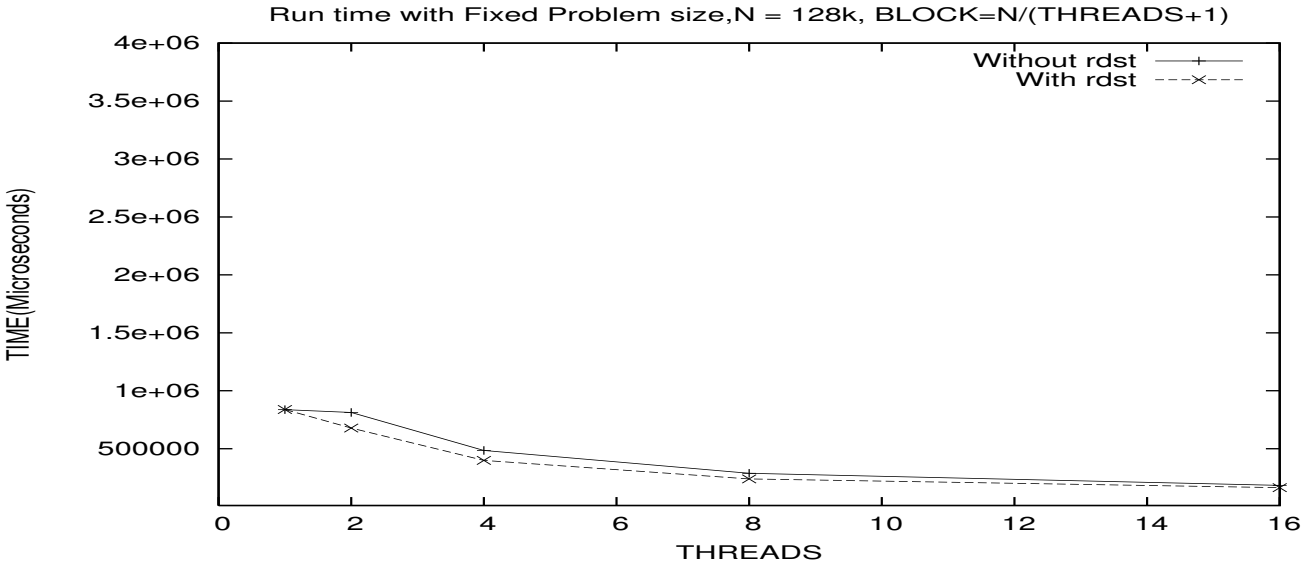

Figure 15. Runtime with BLOCK N/(THREADS+1) (`func`)

6. Summary

The `upc_all_sort` function is an implementation of a generic sort in UPC using a parallel bin sort algorithm. It works in 5 major steps: local sort, bucketize, exchange, merge and rebalance. In each of the steps, locality of source elements is exploited heavily. When a thread needs to access remote elements, it copies those elements to its local shared memory using `upc_memcpy`. This reduces the extra time otherwise needed for multiple remote references. To reduce execution time, threads use local pointers and regular `memcpy` in the local sort and merge operations.

This sort function runs 3.9 times faster than the sequential sort for problem sizes of at least $128K$ with block size `[*]` when run on 8 threads. As the problem sizes and number of threads increases, greater performance is achieved. This sort function runs 6.8 times faster than sequential sort for problem size of $512K$ when run on 16 threads. But the speed up of this function is 3.2 , not as high as 6.8, for problem sizes $32K$ when run on 16 threads. The communication and synchronization overhead in the sort function is responsible for this limitation. The distribution of the source data among threads affects the performance of `upc_all_sort`. When any thread calling this function has no local shared elements the `upc_all_sort` function performs poorly. This performance is improved by choosing the redistribution option.

A. Appendix

Runtime with various block sizes

Figure 16 shows the runtime is same both with and without redistribution for blocksize `[*]`.

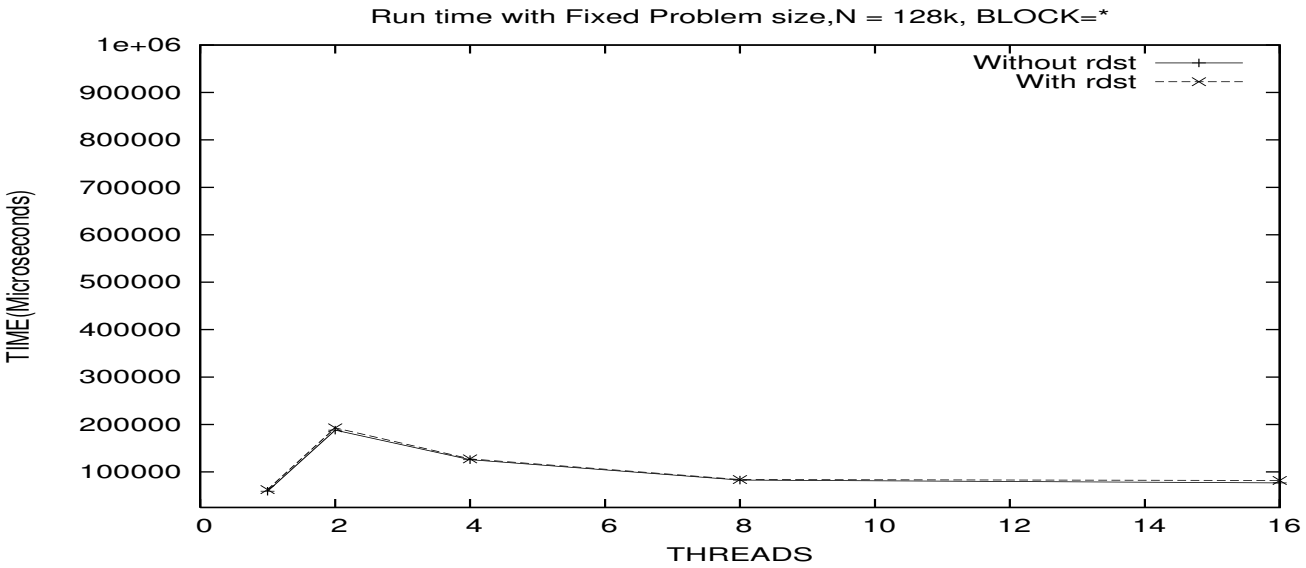

Figure 16. Runtime with BLOCK [*] (hard coded comparison operator)

Figure 17 shows that sorting with redistribution has higher runtime than sorting without redistribution when blocksize is 1. This higher runtime is from the overhead in distribution and reblancing steps.

Figure 18 shows sorting with redistribution has higher runtime than sorting without redistribution for blocksize N. The reason behind this is the overhead in redistribution is still higher than the performance gain for this particular problem size.

Figure 19 shows similar behavior since all the data has affinity to couple of threads.

In Figure 20, the performance with redistribution is simliar with the performance without redistribution as seen in Figure 18, where block size is `N`.

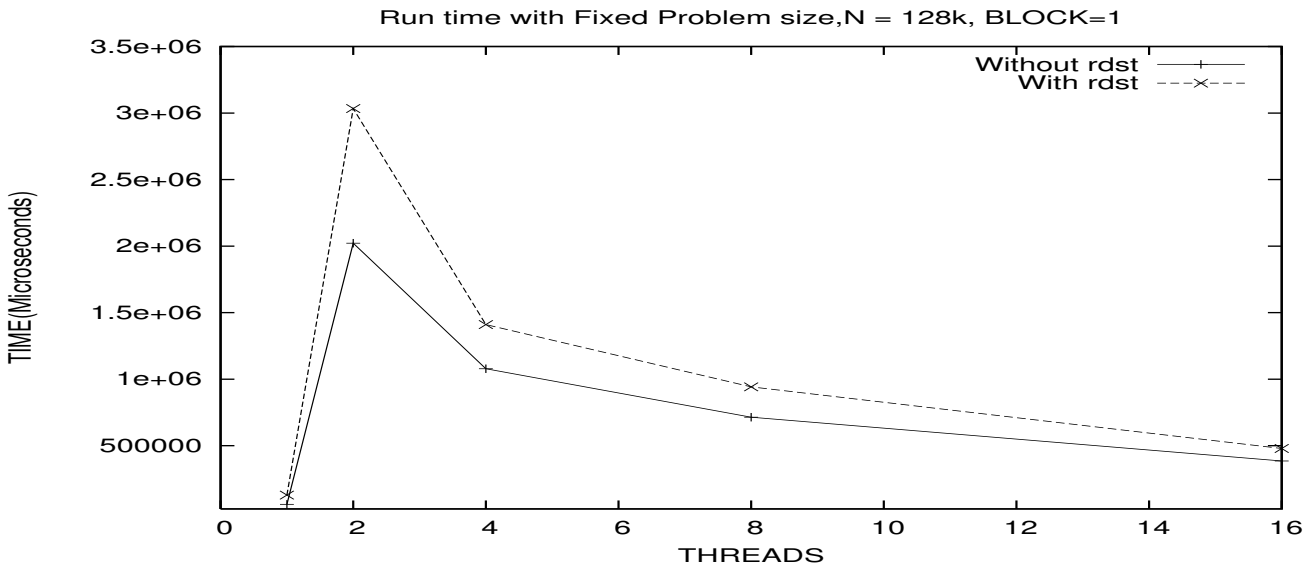

Figure 17. Runtime with BLOCK 1 (hard coded comparison operator)

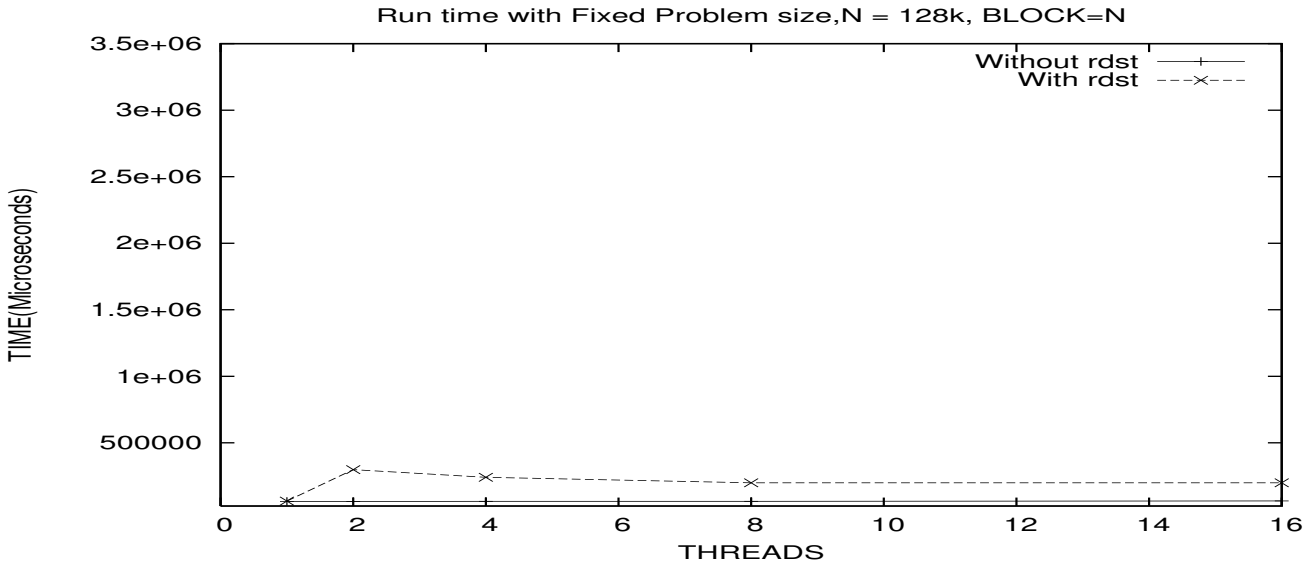

Figure 18. Runtime with BLOCK N (hard coded comparison operator)

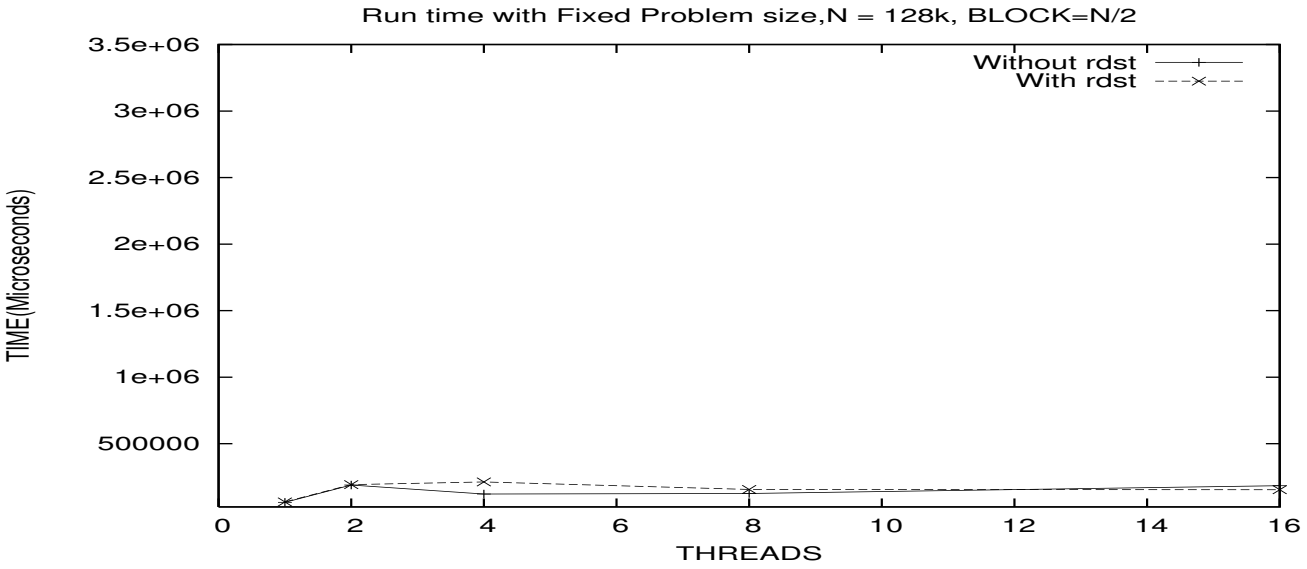

Figure 19. Runtime with BLOCK N/2 (hard coded comparison operator)

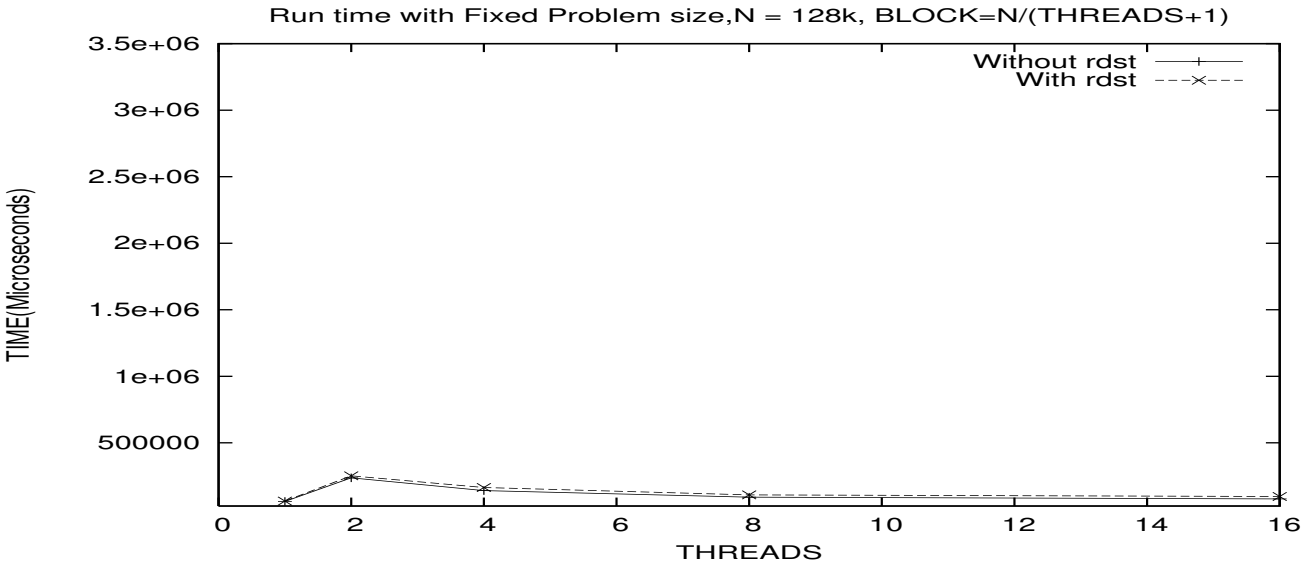

Figure 20. Runtime with BLOCK N/(THREADS+1) (hard coded comparison operator)

References

[1] T. El-Ghazawi, W. Carlson and J. Draper, UPC Language Specification V1.1.1, Technical Report, George Washington University and IDA Center for Computing Sciences, October 7, 2003,<http://www.gwu.edu/~upc/docs/upc_spec_1.1.1.pdf>, March 16, 2005.

[2] E. Weibel, D. Greenberg and S. Seidel, UPC Collective Operations Specifications V1.0, Technical Report, George Washington University and IDA Center for Computing Sciences, December 12, 2003,<http://www.gwu.edu/~upc/docs/UPC_Coll_Spec_V1.0.pdf>, March 16, 2005.

[3] R. Brightwell, J. Brown, Q. Stout and Z. Wen, Experiences implementing sorting algorithms in Unified Parallel C, Sandia National Laboratories, Symposium of the 5-th Workshop on Unified Parallel C. Washington D.C., September 2004.

[4] HP UPC Unified Parallel C (UPC) Programmers' guide, Hewlett-Packard Company, July 2004, <http://h30097.www3.hp.com/upc/upcus.pdf>, March 16, 2005.

[5] S. Seidel, and W. George, Binsorting on Hypercubes with d-Port Communication, in proceedings of the third conference on Hypercube computers and applications, Vol 2, p1455-1461, January 1989.