

Fast Two-Point Correlations of Extremely Large Data Sets

Joshua Dolence¹ and Robert J. Brunner^{1,2}

¹ Department of Astronomy, University of Illinois at Urbana-Champaign,
1002 W Green St, Urbana, IL 61801 USA
`dolence2@astro.uiuc.edu`

² National Center for Supercomputing Applications

Abstract. The two-point correlation function (TPCF) is an important measure of the distribution of a set of points, particularly in astronomy where it is used to measure the clustering of galaxies in the Universe. Current astronomical data sets are sufficiently large to seriously challenge serial algorithms in computing the TPCF, and within the next few years, the size of data sets will have far exceeded the capabilities of serial machines. Therefore, we have developed an efficient parallel algorithm to compute the TPCF of large data sets. We discuss the algorithm, its implementation using MPI and OpenMP, performance, and issues encountered in developing the code for multiple architectures.

1 Introduction

The two-point correlation function (TPCF) is a widely used statistic to quantify the distribution of a set of point processes. In astronomy, it is most often used to measure the large-scale structure of the Universe [7], where whole galaxies are treated as points. Modern astronomical surveys image hundreds of millions and soon billions of galaxies, which necessitates the use of advanced analysis techniques to extract useful scientific information. In addition, large scale simulations of structure formation must be analyzed using these same techniques in order to facilitate comparisons with observations.

A naïve calculation of the TPCF involves $O(N^2)$ distance calculations, since each of the N points must be compared with every other point. Recent algorithmic advances have reduced the complexity of the serial computation to approximately $O(N^{3/2})$, but the calculation remains prohibitive for large N and when the distances of interest span a significant fraction of the data. Worsening the situation, one must also perform the calculations on random data, with sizes of order 50 times as big as the data being examined. Finally, if one wants error estimates more sophisticated than Poisson estimates, the calculation could be slowed by another order of magnitude.

As a result, we have developed a parallelized version which performs efficiently on thousands of processors, bringing the largest current problems and those in the foreseeable future within reach [3]. Below, we briefly discuss the serial algorithm, its parallelization using MPI and OpenMP, code performance,

and a discussion of issues encountered in developing the code for multiple parallel architectures.

2 Serial

The most natural method of computing the TPCF is to divide the distance range of interest into bins, and count the number of data-data pairs (DD) whose separations lie in each bin. These counts, together with similar counts from random-random pairs (RR) and counts of data-random pairs (DR), yields the estimate of the correlation function via an estimator such as Landy-Szalay, which is given by $w(s) = (DD - 2DR + RR)/RR$, where $w(s)$ is the magnitude of the correlation at separation s [5]. The most efficient serial algorithm known attempts to capitalize on this binning by grouping nearby points so that they can potentially be accounted for as a group, rather than individually. It relies on the use of a modified kd-tree, and for reasons that will become apparent, is called the dual-tree algorithm [6].

The kd-tree is constructed top-down, and remains strictly balanced if possible. Each node represents a group of points, with the root node representing the whole data set and its children subsets thereof. The nodes store their spatial (for 3-d data) or angular (for 2-d data) boundaries, the number of contained points, two array indices which uniquely identify these points, and pointers to their children. The nodes are divided recursively until they have fewer than some number of points, at which point they are referred to as leaf nodes.

With the tree(s) constructed, the calculation proceeds by calling a function which recursively scans two kd-trees (just two references to the same tree for DD or RR counting), calculating extremal distances between nodes and determining if they lie within the same distance bin. The biggest savings is realized when large portions of the data can be ignored since they cannot possibly lie within the distances being probed. The algorithm follows, differing slightly from that given originally by [6]:

```
function dualtree(node1, node2) {
  If node1 and node2 have already been compared in reverse
    return
  Let dmin = minimum distance between node1 and node2
  If dmin is greater than maximum distance of interest
    return
  Let dmax = maximum distance between node1 and node2
  Let binmin = distance bin for distance dmin
  Let binmax = distance bin for distance dmax
  If binmin = binmax
    Add node1.cnt x node2.cnt to bin count
  Else if node1 and node2 are leaf nodes
    For all points i owned by node1
      Let smin = minimum distance between point i and node2
      Let smax = maximum distance between point i and node2
```

```

    Let binmin = distance bin for smin
    Let binmax = distance bin for distance smax
    If binmin = binmax
        Add node2.cnt to bin count
    Else
        Find distances to all points in node2 from point i
        Find distance bins and update the bin counts
Else
    If node1.cnt > node2.cnt
        dualtree(node1.leftChild, node2)
        dualtree(node1.rightChild, node2)
    Else
        dualtree(node1, node2.leftChild)
        dualtree(node1, node2.rightChild)
}

```

For a much more detailed discussion of the serial algorithm, including techniques to speed up the various steps in the above routine, the reader is referred to [6] and [3].

3 Parallelization

It is perhaps not immediately obvious, but the dual-tree algorithm is quite amenable to parallelization. A master/slave arrangement is used in which the root compute node responds to requests for work from slave nodes, thereby providing a dynamic load balancing solution. Note that each MPI process has access to all of the data and trees currently being examined and that the two function calls `dualtree(root1.leftChild, root2)` and `dualtree(root1.rightChild, root2)` together are effectively equivalent to the call `dualtree(root1, root2)`. Now define a level L in the binary tree with root `root1` such that there are $W = 2^L$ nodes at that level. Each process then forms an array referencing these W nodes, and when requesting work, receives the index in this array of nodes such that the work it was assigned amounts to `dualtree(nodelist[MyIndex], root2)`, where `nodelist` is the array of nodes and `MyIndex` is the received index. The master process, when not processing messages, performs smaller blocks of work by descending further into the tree. The algorithm for a strictly distributed memory environment is schematically written as follows:

```

function dualtreeMPI(nodelist, root2) {
    Let MyRank = unique process identification
    Let listmax = W
    If master process
        Let listpos = number of processes
        Let mypos = 0
        While listpos < listmax
            Respond to messages, if any, incrementing listpos
}

```

```

    dualtree(nodelist[mypos].leftChild, root2)
    Respond to messages, if any, incrementing listpos
    dualtree(nodelist[mypos].rightChild, root2)
    Set mypos = listpos and increment listpos
Else
    mypos = MyRank
    While mypos < listmax
        dualtree(nodelist[mypos], root2)
        Get next work assignment in mypos
}

```

In, for example, a cluster of multicore systems, the threads sharing memory with the master process access and update `listpos` via OpenMP critical directives. In this case, some care must be taken in multi-threaded MPI calls. In particular, when the MPI implementation does not provide thread support equivalent to `MPI_THREAD_MULTIPLE`, we have found that OpenMP critical directives are required around each communication request in order to ensure proper functionality. Note that on a strictly shared memory architecture, `listpos` is accessed and updated globally so that no master process (and no MPI) is required.

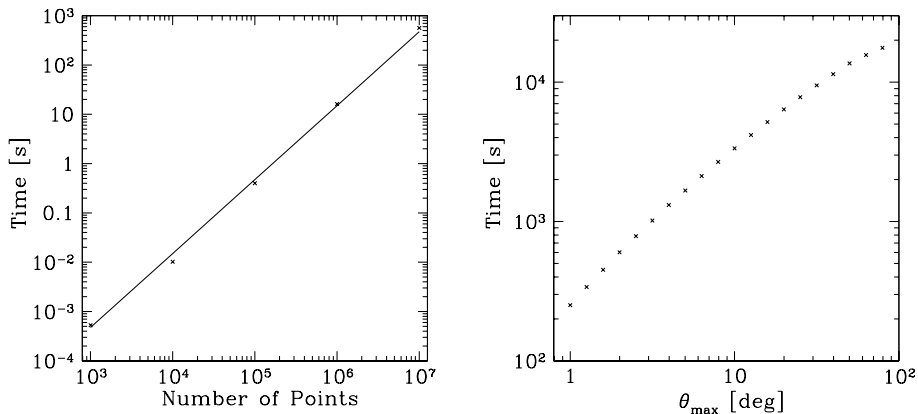
There are two main factors which should influence the value of L , the size of the data set and the number of processors being employed in the calculation. The size of the data set enters by way of giving a maximum value of L . If a leaf node is defined to have N_l points within it or less, and there are N_t points in the entire data set, then the depth of the tree is roughly $D = \log_2 N_t/N_l$. In the algorithm outlined above, we typically have the master process descend two levels further into the tree beyond L so that it is able to check for messages more frequently. Therefore, the maximum value is then $L_{max} = D - 2$. On the other hand, the number of processors sets a minimum value for L . Since there are $W = 2^L$ distinct pieces of work, the minimum value such that every process has some work to do is $L_{min} = \log_2 N_p$, where N_p is the number of processors. Clearly, the best performance is achieved somewhere between L_{min} and L_{max} , and we have found that setting L closer to L_{max} is often desirable.

4 Performance

A variety of tests have been performed to test the algorithm's performance on a range of problems on NCSA's Tungsten and Abe clusters (see table 1). The two main factors controlling the runtime of the code are the number of data points and the maximum separation of interest. The number of bins could, in principle, also affect performance since the bin search operation is performed extensively. However, we have developed a novel adaptive algorithm for this operation which reduces the average number of bins scanned per search to less than one regardless of the number of bins, effectively removing any influence this parameter could have on performance [3]. Figure 1(a) shows the serial scaling of the dual-tree algorithm for an angular correlation with varying numbers of random points

Table 1. Summary of Systems

System	Node Design	Processor	Interconnect	MPI
Tungsten	dual-processor	Intel Xeon 3.2 GHz	Gigabit Ethernet/ Myrinet 2000	ChaMPIon/Pro
Abe	dual socket quad-core	Intel 64 2.33 GHz	Infiniband	MVAPICH2



(a) Runtimes to compute angular bin counts out to 2° . The solid line is a power-law fit with index $3/2$. (b) Scaling with varying maximum angular separation

Fig. 1. Serial scaling results in computing RR bin counts of 10^7 points distributed uniformly over a hemisphere. Calculations were performed on a single processor of NCSA's Tungsten cluster.

distributed uniformly over a hemisphere. Figure 1(b) illustrates the major shortcoming of the serial algorithm, which is poor scaling with increasing maximum separation. In this test, the pair counts of 10^7 points distributed uniformly over a hemisphere were computed with a varying maximum angular separation.

As there is very little communication in the parallel algorithm presented above, it proves to be quite scalable. Figure 2 shows the performance achieved on NCSA's Tungsten and Abe clusters in computing bin counts of the same 10^7 points discussed above, not counting the time to read in and distribute the data. This was done because different systems can have wildly different capabilities with respect to file access and data distribution as will be discussed below. Figure 2(d) shows the Karp-Flatt metric [4]

$$f = \frac{1/s - 1/p}{1 - 1/p},$$

where s is the speedup on p processors, which measures the experimentally determined serial fraction of the program. In all cases, the value of L , described in section 3, is set to 12.

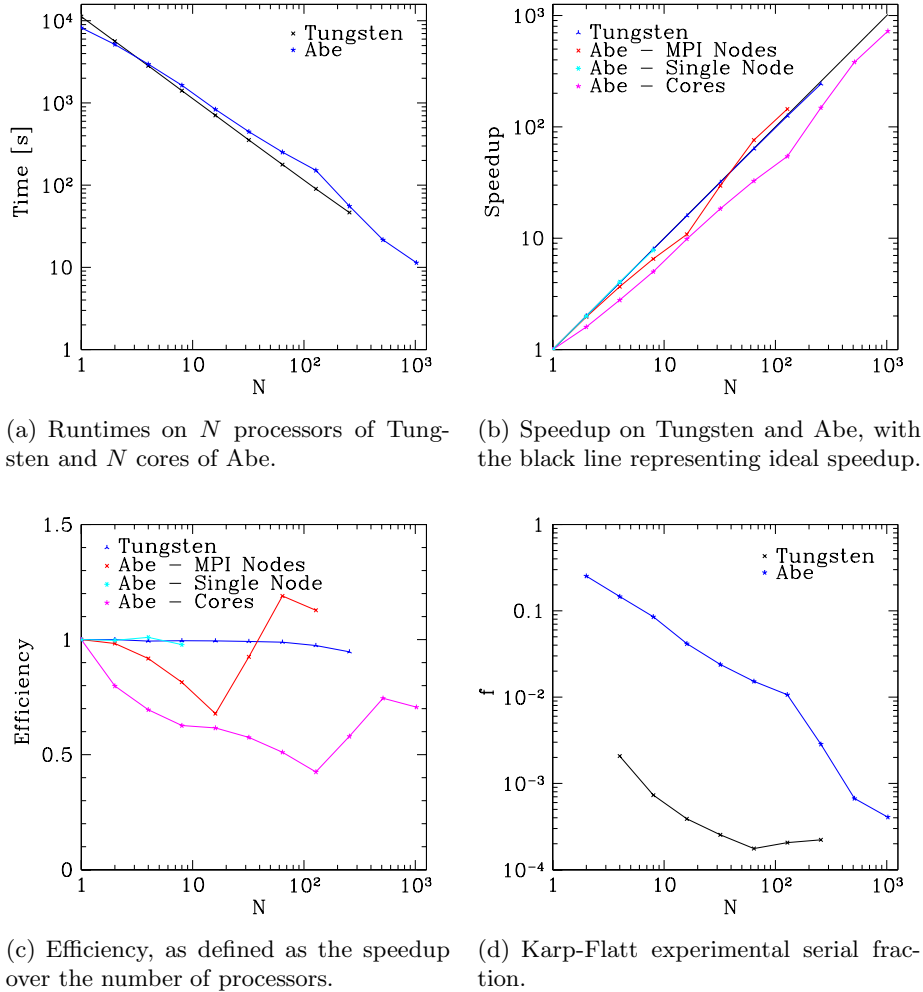


Fig. 2. Parallel performance results on NCSA's Tungsten and Abe clusters. The Tungsten runs utilized MPI only, while on Abe both MPI and OpenMP were used, unless otherwise indicated below. Plots (b) and (c) show multiple lines for Abe which correspond to the following: red points are evaluated considering each 8-core node as a single processor, magenta points consider each node as having 8 processors, and cyan points are the results when the code is compiled with OpenMP support only as described in section 5.2. The calculation considered here is the computation of the RR bin counts out to $\sim 32^\circ$ of 10^7 points distributed uniformly over a hemisphere.

5 Discussion

While figure 2 suggests that the parallel algorithm performs well, two issues in particular can degrade performance on certain systems. First, as mentioned above, is the question of how best to distribute the data. The other issue is particular to multicore systems and is a problem of resource contention.

5.1 Data Distribution

As motivation for a closer look at the problem of data distribution, consider the following very realistic example. One wants to compute the DR counts associated with a data set of 3×10^7 points and a set of 6×10^7 random points on a cluster where each node has 2GB of available memory. These points alone require $\sim 9 \times 10^7 \times 3 \times 8$ bytes, or ~ 2 GB, of memory. Clearly, when the tree associated with each set as well as all of the other (smaller) memory requirements are considered, it is apparent that there is insufficient memory. The solution to this apparent limitation is to break up each set into two or more smaller pieces and analyze pairs of them in turn [3]. The calculation of the DR counts then looks something like

```

For all pieces i of the random set
  Read in randoms(i) and randoms_tree(i)
  For all pieces j of the data set
    Read in data(j) and data_tree(j)
    dualtree(data_tree(j), randoms_tree(i))

```

so that, in the end, it is as if one was able to compute the counts directly. The downside of this approach is that, for a random set broken into N_r pieces, one must read in the entire data set N_r times. Now consider that a full calculation of the correlation function will have ~ 10 such random sets, each needing to be compared with the data as above, so that the data must be read in $\sim 10N_r$ times. A similar analysis holds for computing the DD and RR counts. All told, if the data and randoms cannot fit into memory, a great deal more file access is required than may at first be apparent.

The data distribution issue is most severe for strictly distributed memory systems, since these can necessitate the transfer of large data sets to every processor. For clusters of SMPs, the data must be read only once per shared memory node, so effectively the data is reaching more processors per transfer. Finally, strictly shared memory systems require only a single thread to read in the data, effectively removing any effect the data distribution has on performance. For the purposes of this study, we have so far focused on performance on NCSA's Tungsten and Abe clusters. Given that one would expect lower performance from Tungsten due to the above argument, and that Abe nodes are reportedly capable of reading data at ~ 10 times the rate of Tungsten nodes [1], we have focused the proceeding tests solely on Tungsten.

There are several alternative techniques one could employ to globally distribute a data set. Assuming all processes have file access, one solution would

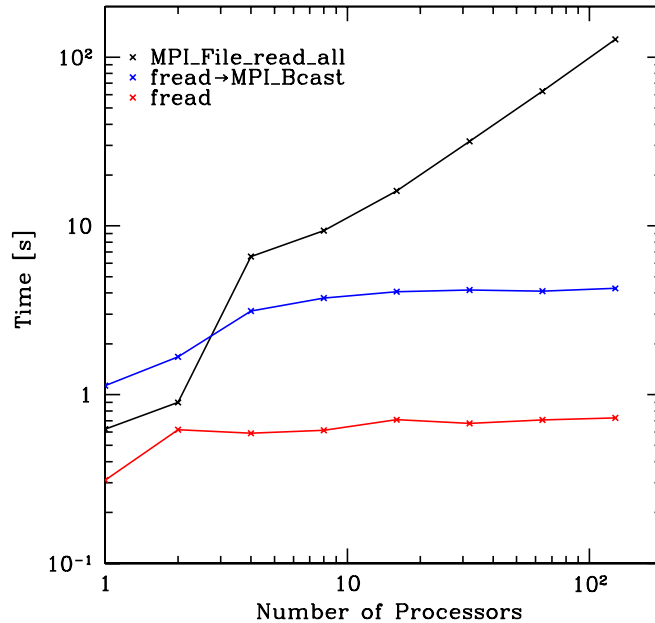


Fig. 3. The time to read in a data file containing 10^7 points ($\sim 229\text{MB}$) on NCSA's Tungsten cluster using various techniques.

be to simply have each process read the data independently. Another obvious solution is to have one process read in the data and then broadcast it to all other processes. One could also utilize the file access functions defined in the MPI-2 standard, and rely on their efficient implementation. Lastly, if each compute node has local disk space, it is possible to copy the data files to each compute node and have each process read from its local disk.

The performance of all but the last of these options on Tungsten is shown in figure 3. For this test, a data set containing 10^7 points stored on Tungsten's LUSTRE filesystem without striping was distributed to each processor. In particular, reading in a data file amounts to three separate read calls, the first two to read in header information which amounts to ~ 20 4-byte integers, and the last reads in 3×10^7 8-byte doubles (the factor of 3 comes from the use of Cartesian x , y , and z coordinates). Quite surprisingly, the independent `fread` commands significantly outperformed the other methods. In fact, the `MPI_File_read_all` approach, which arguably should have performed the best with a large number of processors, seems to be serializing the file accesses so that only a single processor is actively receiving data at any given time. It is difficult to imagine a less efficient approach, and shows how important it is to test key features rather than relying on their efficient implementation.

With disk striping on the LUSTRE file system enabled, the performance of `MPI_File_read_all` improves, as would be expected, while the other techniques retain the same rate of throughput. However, we have found that the improvement still does not reconcile the difference between this and the simple `fread` approach. For example, with the data file striped across eight Object Storage Targets with a stripe size of 65536 bytes, the read time on eight processors was 4.2 seconds. Comparing this with the results shown in figure 3, we see that it is still slower than both the `fread` and `fread`→`MPI_Bcast` approaches.

On the other hand, the issue of portability may not favor the independent `fread` approach. In the first place, it requires that every MPI process has access to the file system which, thankfully, is now quite common. However, for good performance, each process must be able to read from a file *concurrently* and the interconnect between the file system and compute nodes must be at least comparable to that connecting compute nodes to one another. If either of these is not satisfied, this approach is likely to perform badly.

Although thus far untested, the idea of copying the files to node local disks is likely to be beneficial in certain situations. In general, this would involve using one of the above techniques followed by writing out the data set(s) to a local file(s). Future access to these data sets would then be accomplished locally. Therefore, one would naturally expect performance to be improved only if the file must be read in multiple times, the possibility of which was described at the beginning of this section.

5.2 Multicore Systems

The issue of resource contention on multicore systems is widely known. In particular, the cores typically share the same system bus and have only a limited memory bandwidth and so memory intensive applications often do not scale well to a large number of cores [2].

Tests on a node of NCSA's Abe cluster, when compiled using Intel's C compiler without MPI, show that the dual-tree parallelization achieves a speedup (on 8 cores) of approximately 7.5, or about 94% efficiency. While there apparently is some performance degradation due to resource contention, the problem is clearly not severe, at least on this system. However, when the code is compiled with `mpicc -cc=icc`, the single node performance is drastically reduced, achieving a speedup (on 8 cores) of only 5 so that the efficiency is only about 63%. At the time of this writing, it is unclear why such a discrepancy exists as when only a single MPI process is used, the code reverts to the shared memory algorithm which performs so well. As shown in figure 2(c), the loss of efficiency on a per core basis is essentially entirely accounted for by this poor single node performance. Therefore, if this issue can be resolved, we expect the magenta points in figure 2(c) to change for the better.

6 Conclusions

The TPCF is widely used both in astronomy as well as in other disciplines. Traditional methods of computing the TPCF are grossly inadequate for the massive data sets available today, and will become more so in years to come. Recent algorithmic advances have improved the situation, but the calculation remains prohibitive for the largest of these data sets and when one is interested in structure at scales that are a significant fraction of the physical size of the data set.

Therefore, the utilization of high-performance computing resources is a vital next step. Here, we have presented a parallel algorithm which performs efficiently in cluster and multicore environments and discussed some of the issues encountered in its development. The issues of how best to distribute the data and improving the hybrid MPI/OpenMP performance are still being actively investigated, and may ultimately be system dependent.

Acknowledgements The authors acknowledge support from NASA through grant NN6066H156, from Microsoft Research, and from the University of Illinois. The authors made extensive use of the storage and computing facilities at the National Center for Supercomputing Applications and thank the technical staff for their assistance in enabling this work. The authors also wish to thank the reviewers for their helpful suggestions to improve the presentation and technical content of this paper.

References

1. Arnold, G.: private communication
2. Chandra et al.: Parallel Programming with OpenMP. Academic Press (2001)
3. Dolence, J., Brunner, R. J.: (2008) in prep.
4. Karp, A. H., Flatt, H. P.: Measuring Parallel Processor Performance. Communications of the ACM **33** (1990) 539–543
5. Landy, S. D., Szalay, A. S.: Bias and Variance of Angular Correlation Functions. Astrophysical Journal **412** (1993) 64–71
6. Moore, A., et al.: Fast Algorithms and Efficient Statistics: N-Point Correlation Functions. Mining the Sky: Proceedings of the MPA/ESO/MPE Workshop (2001) 71
7. Peebles, P. J. E.: The Large-Scale Structure of the Universe. Princeton University Press (1980)