2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems

A fast, Scalable SLINK Algorithm for Commodity Cluster Computing Exploiting Spatial Locality

Poonam Goyal, Sonal Kumari, Sumit Sharma, Dhruv Kumar, Vivek Kishore, Sundar Balasubramaniam, Navneet Goyal
Advanced Data Analytics & Parallel Technologies Laboratory
Department of Computer Science & Information Systems, BITS-Pilani, Pilani Campus, INDIA

***Abstract*—Single linkage (SLINK) hierarchical clustering algorithm is a preferred clustering algorithm over traditional partitioning-based clustering as it does not require the number of clusters as input. But, due to its high time complexity and inherent data dependencies, it does not scale well for large datasets. To the best of our knowledge, all existing parallel SLINK algorithms are based on the traditional SLINK algorithm and thus require large number of computing resources. In this paper, we present a novel optimization of SLINK algorithm, *Grid*SLINK, which is an order of magnitude faster than the existing state-of-the-art implementation. The optimization in *Grid*SLINK comes from reduction in number of distance calculations required by SLINK. This reduction is achieved by exploiting spatial locality of data points and using an adaptive gridding technique. *Grid*SLINK is parallelized for distributed memory systems. Scalable performance is achieved for increasing number of compute nodes. The proposed parallel algorithm, *dGrid*SLINK, is benchmarked against the best existing parallel algorithm in literature and found to outperform the latter for all the real datasets considered. *dGrid*SLINK can cluster millions of data points in few seconds/minutes using a small number of processing elements, without compromising the quality of clustering.**

Keywords—Parallel computing; multi-core; multi-node; clustering; SLINK; adaptive gridding

I. INTRODUCTION

Hierarchical clustering [1] creates a series of nested partitions which can be intuitively visualized in the form of a tree called a dendrogram. The main advantage of hierarchical clustering over partition/representation based clustering is that the number of clusters required, need not be given as input. The number of clusters can be decided either at the end of the iterative algorithm or after some iterations. Moreover, there is no need to re run the algorithm if we need more or less number of clusters unlike, density-based algorithms, which require re running the algorithm, with changed input parameters. Hierarchical agglomerative clustering (HAC) algorithms start with each data point as a cluster and keep merging closest clusters iteratively until we get a single cluster having all points. The desired number of clusters can be obtained by cutting the dendrogram at an appropriate level. A number of similarity (or dissimilarity) measures have been used for HAC [1]. Single-linkage (SLINK), Complete linkage (CLINK), Average linkage, Centroid-based HAC, Median-based HAC, Minimum variance Ward's method are some HAC algorithms which essentially differ in the similarity measures used.

SLINK [2], one of the most popular HAC algorithms, is similar to Minimum Spanning Tree (MST) problem [3]. An MST can be seen as cluster hierarchy (dendrogram) and therefore SLINK requires the same computational complexity as that of the MST problem. The Sibson's algorithm is the state-of-the-art solution for SLINK [2]. It requires $O(n^2)$ time and computes the distance of every point with all the other points to arrive at the final dendrogram and does not take into account *spatial locality* of data. S*patial locality* of a data point gives the neighborhood information about those data points which can be utilized for efficient neighborhood computation. In SLINK, the most similar pair of clusters is the pair with the minimum inter-cluster distance which is the minimum distance between any pair of points in the two clusters. This implies that the probability of a point merging with a point in its neighbourhood is higher than with a point which is far away from it. It may happen that all the points are merged into a single cluster by considering the points in their *immediate neighbourhoods* only. This immediate neighbourhood of any point can be defined as the smallest neighbourhood with the help of which we can obtain the final dendrogram, without requiring to compute distances between points that are far. Therefore, we need to consider only a small fraction of all pairs of points for distance computations and merging. An efficient implementation of SLINK, *Grid*SLINK, is proposed which runs significantly faster than that of SLINK in [2] and produces clustering which is exactly the same as obtained by SLINK.

*Grid*SLINK works on cells obtained by gridding the data space. Initially, the data space is partitioned into a number of equal-size grid cells. The number of data points vary in each cell and this variation depends on the data distribution. For skewed data, in which density variation is too high, the load balancing is achieved and locality is preserved by further dividing the cells along each of the d dimensions resulting in 2^d sub-cells. This results in approximately equi-depth cells of varying sizes. This technique is referred to as *adaptive gridding* [4], [5]. *Grid*SLINK works on cells resulting from adaptive gridding to achieve efficient region queries and to create cell MSTs containing all points in a cell. The local MSTs are then merged using an approach similar to Kruskal's algorithm [6] and the *union-find* data structure [7] to get the final MST or dendrogram.

Hierarchical clustering is challenging to parallelize due to its high data dependency, i.e., each level of merging depends on all previous merges. Some earlier parallel implementations [8]–[10] stores distance matrix and need to communicate large amount of data to update similarity matrix and thus cannot handle large datasets. Recently, researchers have parallelized SLINK [2] using disjoint data structure [11]–[14]. These parallel algorithms are scalable, but are computationally expensive.

They require a large number of compute nodes to cluster data in reasonable amount of time.

We present a scalable parallel algorithm for *Grid*SLINK, called *dGrid*SLINK, for distributed memory architectures. *dGrid*SLINK is compared against PINK [12] which is the best known existing distributed memory implementation of SLINK. PINK performs the same number of computations as classical SLINK, but it uses $\lceil k^2/2 \rceil$ processors for computing cross-edges (from points at a node to points at another node) and self-edges, where $k \geq 1$. The number of processors is dependent on data partitions (k) and it increases exponentially with k.

*dGrid*SLINK is up to 208.62x faster than PINK on 18 nodes for data of size 3M. The efficiency of *dGrid*SLINK can be attributed to the fact that locality-based cell MST merging reduces a large number of cross-edges (from points in a cell to points in other cell) computations. Another way of looking at it is that if PINK has to finish an execution in same amount of time as *dGrid*SLINK, then it has to use 3756 nodes as compared to 18 nodes by *dGrid*SLINK to cluster 3M data points. Experiments have been performed on several data sets of size varying from 3M to 57M, and dimensionality varying from 3 to 10.

*dGrid*SLINK ensures load-balancing by a novel technique for spatially distributing equal amounts of data to multiple nodes. At each node, cell MSTs are computed for each local cell in a data-parallel mode. Cell MSTs are merged iteratively using neighboring cells to update global MST. The algorithm converges when a single MST is formed. In *dGrid*SLINK, the data distribution among nodes is done sequentially.

The rest of the paper is organized as follows. Section II discusses details of related work. The proposed sequential and parallel algorithms are given in Section III and IV, respectively. Experimental set up, dataset specifications, and result analysis are reported in section V. Section VI concludes the work presented in the paper and gives future directions.

II. Related Work

SLINK is one of the popular HAC algorithms starting with each point as a cluster and then repeatedly merging the clusters to get a single cluster. It has time and space complexity O(n^2). Its most efficient implementation is given by Sibson [2]. Sibson reduced the space complexity from O(n^2) to O(n) by only storing single row of the distance matrix at a time.

Similarity matrix based parallel algorithms are presented in [8]–[10]. Li [8] presented a parallel hierarchical algorithm having O(n^2) time complexity, for single instruction multiple data model which uses shuffle exchange network to access similarity matrix and input data. Wu et al. [9] presented a parallel hierarchical algorithm using reconfigurable optical buses (AROB) architecture. The limitations of [8], [9] are that they are designed for special parallel architectures. In 2005, Du and Lin [10] used Message Passing Interface (MPI) [15] to present a distributed hierarchical clustering algorithm. Similarity matrix along with the data points is distributed across multiple nodes and then synchronized at each merging step. The clustering quality is dependent on the chosen input parameter threshold. Similarity matrix based parallel algorithms do not scale well due to high communication cost.

SLINK problem can be seen as an MST problem [3]. Therefore, many researchers have attempted SLINK as constructing MST in parallel. Olson [16] gave two parallel solutions for SLINK specific to shared memory PRAM model and to distributed memory parallel machine with butterfly architecture. Bentley [17] gave a distributed memory algorithm for constructing MST by assigning computations to a number of 'tree' processors. However, the work reported in [16], [17] gave theoretical analysis but are not evaluated emperically.

Olman et al. gave a distributed memory parallel clustering algorithm, CLUMP [18]. They have considered whole data as graph which is partitioned randomly into smaller sub-graphs which is composed of complete bipartite graphs, then compute MST for each sub-graph. They compute MSTs for self-edges and cross-edges by distributing the load among multiple nodes. These local MSTs are merged to get the final MST. The basic idea is to minimize the communication cost at the expense of redundant computations. Hendrix et al. present PINK [12], similar to CLUMP algorithm. They also minimize the communication cost by decomposing the problem into sub-problems, which removes redundant computations at the same time. However, partitioning the whole data into k partitions and assign each ${}^{k}C_2$ possible combinations to various nodes for cross-edge and self-edge computations. Overall, they need $\lceil k^2/2 \rceil$ processors for computing cross-edges and self-edges, where $k \geq 1$. The major drawback of their technique is that, for increasing k (and so p), redundant data will keep on increasing exponentially which may affect the scalability of the algorithm. Similar algorithms [13], [14] are designed for MapReduce and Spark frameworks, respectively and suffer from same limitations. A similar MapReduce based parallel but incremental approach [19] is also reported in the literature.

Johnson and Kargupta [20] present an approximate parallel SLINK algorithm for distributed memory systems. They have used lower and upper bounds for the distances between any two given points to minimize the data communication. The upper bound refers maximum possible distance between two points in the dendrogram and the lower bound refers the distance value stored in the lowest root of the subtree connecting the two leaves in the dendrogram. Using these bounds local dendrograms are merged to get global dendrogram. However, authors did not validate scalability of the algorithm.

There are several parallel algorithms [11], [16], [21]–[24] given for shared memory systems. Rajasekaran [23] gave an efficient parallel hierarchical algorithm for PRAM shared memory systems and AROB distributed memory systems. This algorithm partitions the data into grids of uniform size under the assumption that data points are uniformly distributed within a unit square in the plane. But, this assumption is not always true. The algorithm pPOP [25] is a parallel version of the POP algorithm given by Dash et al. [3]. POP is a HAC algorithm which runs in two phases. In the first phase, data is partitioned into p overlapping cells. In each iteration, the closest pair in each cell is found to obtain the overall closest pair. If the overall closest pair is less than an overlap threshold, δ, those pairs are merged. If the distance of the closest pair exceeds δ, phase 2 performs hierarchical clustering. The authors claim that 90% of merging is performed in phase 1 and the last 10% of merging takes negligible time. In pPOP, phase 1 is parallelized and phase

2 is run sequentially as it takes negligible time. This is reasonable for all centroid/medoid based algorithms, but not for algorithms using graph metrics, such as SLINK.

All the existing algorithms are at least performing n^2 computations and hence inefficient for large data sets. This leads to use of large number of resources to complete a task in reasonable amount of time. *Grid*SLINK and *dGrid*SLINK use spatial locality by considering distance computation with only those points that are in the immediate neighborhood, resulting in reduced number of distance computations. Therefore, the proposed implementations achieve significant performance improvements over counter parts.

III. The proposed sequential SLINK: *Grid*SLINK

*Grid*SLINK, is an MST-based implementation of SLINK that uses the concept of grids. It involves partitioning of data space into cells using *adaptive gridding*, computation of cell MSTs, and merging them to get a global MST.

Initially, the data space is divided into multiple cells of equal length, $CellSize_{init}$, along each dimension. The maximum number of distance computations required for local MSTs is $\Sigma_j CCount_j^2$, where j is the cell identifier. *CCount* is defined as total number of points present in a cell. The number of distance calculations is minimum when *CCount* values are uniform across cells. But this cannot be expected for most datasets. The adaptive gridding process is used to keep the *CCount* values within a small range in order to ensure load balancing. Adaptive gridding [4] divides each *dense* cell into 2^d sub-cells, where d is the number of dimensions. A cell is referred as a dense cell if its $CCount > \tau$, a threshold. The process of dividing dense cells is repeated until no dense cell remains. The $CellSize_{init}$ is calculated as $\sqrt[d]{(RegionSize * \tau)/n}$, where *RegionSize* is the volume of the data-space and n is the total number of points in the dataset. Fig. 1 presents a snapshot of the adaptive gridding process performed over a two-dimensional data-space. The parameters, $CellSize_{init}$ and *CellSize,* denote the cell-size of the biggest cell and the smallest cell.

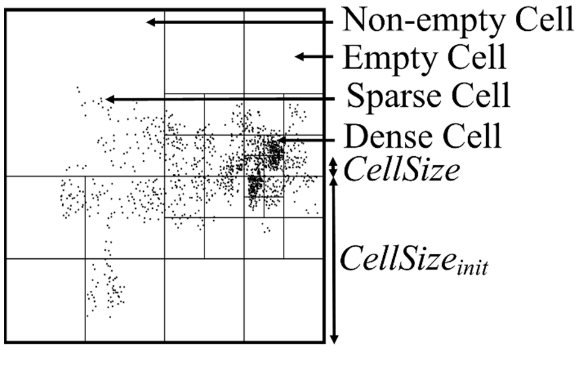

Fig. 1. Adaptive gridding

Two data structures, *R*-tree [24] and *CellList* are used to perform adaptive gridding. Cells are stored in *R*-tree as data objects and points are added in the corresponding cells in the *CellList*. Fig. 2 depicts the connection between these data structures. *R*-tree is constructed by inserting points one by one. To insert a point in the data structure, first the coordinates of the cell that would contain this point after gridding is obtained and a region query for this cell is fired on the *R*-tree. The point is added to the cell if the cell already exists otherwise new cell with these coordinates is constructed and inserted in the *R*-tree and *CellList*. Adaptive gridding is applied on dense cells. For this, the dense cell is removed from the *R*-tree and the *CellList* and its sub-cells (obtained by dividing it) are added into the *R*-tree in the same manner and are also appended to the *CellList* for their efficient enumeration. The number of elements in the *R*-tree equals to the number of cells which is approximately n/τ, reducing the processing time of region queries in *R*-tree. A detailed analysis of execution of region queries on *R*-trees can be found in Faloutsos et al. [26].

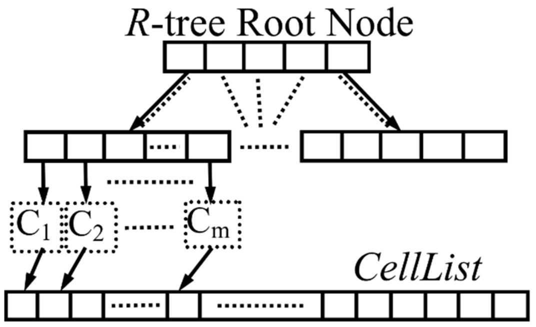

Fig. 2. *R*-tree and *CellList*

```
input: DataList S, number of points n, threshold points per cell τ
output: global MST gmst
1: procedure GridSLINK(S)
      //R-tree rt; CellList CL; EdgeLists EL
2:   (rt, CL) ← constructRTreeAdaptive(S, τ)
3:   for each unprocessed cell C in CL
4:      (C.mst, EL) ← constructCellMST(C)
5:   Initialize uf;  // the union-find data structure
6:   gmst ← NULL
7:   $\mathcal{E}$ ← CellSize; $\mathcal{E}_{prev}$ ← 0; itr ←1
8:   while gmst has < n - 1 edges
9:      EL ← removeRedundantEdges(EL)
10:     for each cell C in CL
11:        EL ← computeCrossEdges(C, $\mathcal{E}$, $\mathcal{E}_{prev}$, rt, itr)
12:     (gmst, uf) ← updateglobalMST(EL)
        //edges of length w added, where $\mathcal{E}_{prev} \le w < \mathcal{E}$
13:     $\mathcal{E}_{prev}$ ← $\mathcal{E}$; $\mathcal{E}$ ← $\mathcal{E}$ + CellSize *($2^{itr}$);   itr++
```

Fig. 3. The proposed *Grid*SLINK algorithm

Prim's algorithm [27] is applied to each cell to construct cell MSTs of completely connected graph of points. This essentially clusters points in each cell using SLINK. The *union-find* data structure [7] is used for determining membership of a point. It is a rooted tree whose root denotes parent of a data point x exist in a cluster. Initially the root of each point is set to be the point itself (i.e. singleton tree). It performs two operations, find and union, where *find* is same as search operation in a tree. The operation *union* merges two points x and y *by assigning* y_{root} as child of x_{root}, if x_{root} is greater than y_{root} or vice-versa.

Edges with weight w (Euclidean distance between its end points) of the local MSTs are stored in *EdgeLists* data structure. *EdgeLists* is an array of lists, such that j^{th} list contains edges whose weight is w, where $(2^{j-1} - 1)*CellSize \le w < (2^j -$

1)**CellSize,* for $j > 0$. *Grid*SLINK is an iterative algorithm, the pseudo code is given in Fig. 3. A Kruskal like algorithm is used to construct global MST in each iteration requiring *cross-edges* computation. The process stops when n-1 edges have been added to the global MST. *Cross-edges* computation and global MST updation are done as follows:

For each cell C, in the *CellList*:

Compute the cross-edges between points in C and points in those cells which lie in an annular region (called neighboring cells) obtained as follows:

extend the boundaries of C by $\mathcal{E}$ along each dimension, but exclude the region obtained by extending the boundaries of C by $\mathcal{E}_{prev}$ along each dimension (see Fig. 4) *where initial $\mathcal{E}$ = CellSize and $\mathcal{E}_{prev} = 0$. $\mathcal{E}$ and $\mathcal{E}_{prev}$ are updated in each iteration as $\mathcal{E}_{prev} = \mathcal{E}$ and $\mathcal{E} = \mathcal{E}$ + CellSize *(2^{itr}) where itr = iteration number* ($1 \leq itr \leq$ constant).

If all points in C and all points in a neighbouring cell belong to the same cluster, then there is no *cross-edge* to be computed between them. To avoid duplicate distance computations for *cross-edges,* only half of the annular region is considered (see Fig. 4).

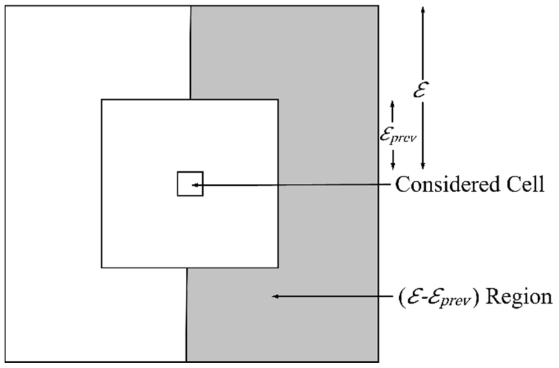

Fig. 4. Region considered for cross-edge computation for an itr ≥ 2

For all those points in C that belong to a single cluster s_1 and all those points in a neighbouring cell that belong to another single cluster s_2, the minimum weight *cross-edge* between s_1 and s_2, is added to the *EdgeLists*. Only those edges whose weight are in between $\mathcal{E}_{prev}$ and $\mathcal{E}$ in the *EdgeLists* such that its endpoints are in two different clusters are considered for addition to the global MST. These edges are considered in non-decreasing order of weights to ensure that the minimum among the *cross-edges* belonging to different clusters is the one that gets added to the global MST. The number of lists sorted per iteration is one because only the edges with weight between $\mathcal{E}_{prev}$ and $\mathcal{E}$ will contribute to the global MST and are stored in a single *EdgeList*. After updating gmst, the edges from current *EdgeList* are deleted and redundant edges are also removed from the *EdgeLists*. Redundant edges are those edges which make cycles and will never contribute to gmst.

Complexity Analysis:

The runtime complexity of *Grid*SLINK is derived step-wise and given in Table I using *numC* as number of cells, m as min-entries in R-tree, $|E_L|$ as the maximum size of a local *EdgeList*, *iter* as number of iterations and α as the inverse of Ackermann's function.

TABLE I. TIME COMPLEXITY OF *dGrid*SLINK

S. No.	Steps	Time complexity
1.	R-tree construction	$O(n * log_m\, numC)$
2.	Cell MST computation	$\left(\left(\frac{n}{numC}\right)^2 * numC\right) = O\left(\frac{n^2}{numC}\right)$
3.	Cross-edges	$O(n^2)$ because each cross-edge is unique and computed only once across all iterations; and the total no. of cross edges are bounded by n^2
4.	*EdgeList* sorting	$O(iter * \|E_L\| * log\|E_L\|)$ Since in each iteration, the annular region grows exponentially, from *CellSize* to *RegionSize*. So, $iter = log_{CellSize}\, RegionSize$ and in each iteration we add at most $n * \left(\frac{\mathcal{E}-\mathcal{E}_{prev}}{CellSize}\right)^d$ *cross-edges,* which is $O(n)$ because $\left(\frac{\mathcal{E}-\mathcal{E}_{prev}}{CellSize}\right)$ is a constant. $= O(\log RegionSize * n * \log n)$
5.	Global MST update	$(iter * \|E_L\| * \alpha(\|E_L\|)) = O(iter * n * \alpha(n)) = O(\log RegionSize * n * \alpha(n))$
6.	Total	$O(n^2)$

Iter is small and depends on τ. But, it does not get affected if τ is chosen appropriately (see Fig. 8 and its discussion).

Scalability Analysis:

*Grid*SLINK produces correct clustering and having time complexity of $O(n^2)$. *Grid*SLINK is much faster than classical SLINK. *Grid*SLINK can achieve close to *linear speedup* as follows:

$$\text{Effective speedup} \geq \frac{n*p}{(p-1)*(log_{CellSize}\, RegionSize)*\,\alpha(n) + n} \geq \frac{p}{p*\beta(n) + 1}$$

, where $\beta(n) = O(\frac{\alpha(n)}{n})$ is a decreasingly small value and α is the inverse of Ackermann's function.

Experimental results of our parallel implementation of *Grid*SLINK validate the effective speedup estimation (given in Section V).

IV. *DISTRIBUTED MEMORY Grid*SLINK*: dGrid*SLINK

*dGrid*SLINK parallelizes *Grid*SLINK for distributed memory multi-node systems. The algorithm can be divided into three key steps: 1) data distribution, 2) local computations, and 3) update global MST to get final dendrogram. Fig. 6 shows the control flow diagram of *dGrid*SLINK. The pseudocode of *dGrid*SLINK is given in Fig. 7.

1) Data Distribution

Spatial data distribution is performed using gridding to preserve spatial locality. Data space is first divided into k_1 equal partitions by gridding along one dimension and then each partition is further divided into k_2 equal partitions along second dimension and so on until we get $p = k_1 \times k_2 \times ... k_l$ partitions where l is the l^{th} dimension (see Fig. 5). For example, if we take $k_1 = k_2 = 32$, then we will get 1024 partitions for $l = 2$. l will always be very few in number. We select l dimensions in decreasing order of maximum span of dimensions to split successively. We stop splitting along one direction when further division is not possible i.e. only one cell remains in the strips. The number of partitions along a dimension is decided based on the initial size of the cells i.e. $CellSize_{init}/r$, where r (≥1) is a constant. We have divided *CellSize$_{init}$* by *r* to get better load balancing, as initial size of the cell is large. Partition boundaries are aligned to cell boundaries in all cases.

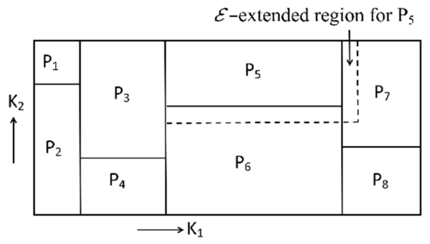

Fig. 5. Data distribution in *dGrid*SLINK algorithm for k_1=4, k_2=2 and l=2

2) Local Computation

Each node, constructs *R*-tree (R_0), Local *CellList* (CL_0) and performs adaptive gridding on local data. *CellSize* denotes the global smallest edge-length of any cell in the CL_0 across the nodes. Each node maintains its *local EdgeLists* (*EL*) to store the local edges. Cell MSTs are computed for every cell at a node. Cross-edges for each cell are computed iteratively with other cells in its annular region. For cross-edge computation, remote data in $\mathcal{E}$-extended region from partition boundary is required (see Fig. 5). $\mathcal{E}$-extended data contains annular region required for each cell cross-edge computation in an iteration. We send only the additional data in an iteration to avoid redundant data communication. This strategy of data distribution helps in minimizing communication cost for cross-edge computations. We communicate data along one dimension up to $p=k_1$ and then along two dimensions for $p=k_1 \times k_2$ and so on. Fig. 5 presents initial data distribution and data communication during iterations.

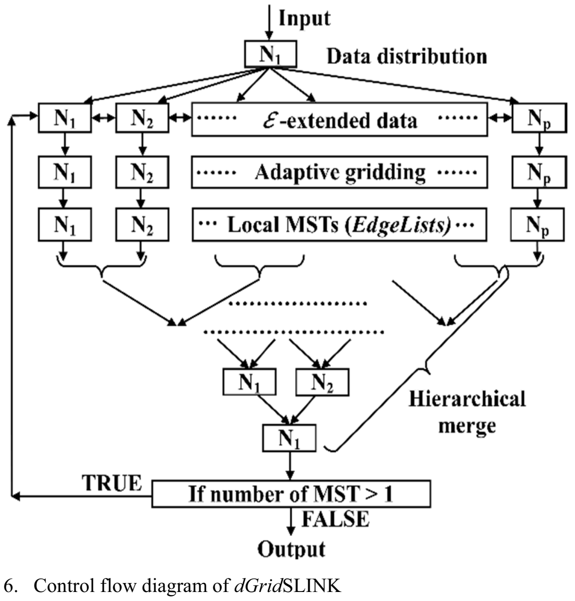

Fig. 6. Control flow diagram of *dGrid*SLINK

We also construct *R*-tree (R_{itr}) and local *CellList* (CL_{itr}) for $\mathcal{E}$ -extended data at each iteration at every node for efficient neighborhood queries. We compute cross-edges for each cell in CL_0 at every iteration and add them into its *EL*. We also keep a local *union-find* data structure to determine the membership of a point and store local MSTs.

Fig. 7. The proposed *dGrid*SLINK algorithm

3) Update global MST to get final dendrogram

```
input: DataList S, number of points n, $\tau$, number of processes p
output: Global MST gmst
1: procedure dGridSLINK(S)
     //$DL_{itr}$ and $CL_{itr}$ are local data list and respective cellist
2:   for process $P_1$
3:     dataPartition(S, p) // results in p local data lists
4:   forall process $P_i$, $1 \le i \le p$
5:     ($rt_0$, $CL_0$) = constructRTreeAdaptive($DL_0$, $\tau$)
6:     update the minimum of CellSize
7:     for each C in $CL_0$
8:       (C.mst, EL) $\leftarrow$ constructCellMST(C)//EL is local EdgeLists
9:     mst $\leftarrow$ NULL; gmst $\leftarrow$ NULL
10:    initialize uf & guf; //uf and guf are local and global union-find data
         structures resp.
11:    $\mathcal{E} \leftarrow$ CellSize; $\mathcal{E}_{prev} \leftarrow 0$; itr = 1; $\delta \leftarrow \mathcal{E}$
     //end forall
12: while gmst has < n - 1 edges
13:    forall process $P_i$, $1 \le i \le p$
14:      $\delta \leftarrow \mathcal{E} - \mathcal{E}_{prev}$ //$\delta \le \mathcal{E}$
15:      get $\delta$-extended data from $P_l$
16:      ($rt_{itr}$, $CL_{itr}$) = constructRTreeAdaptive($DL_{itr}$, $\tau$)
17:      EL $\leftarrow$ removeRedundantEdges($EL_i$);
18:      for each C in $CL_0$
19:        EL $\leftarrow$ computeCrossEdges(C, $\mathcal{E}$, $\mathcal{E}_{prev}$, $rt_0$, itr)
20:        for each j, $1 \le j \le$ itr
21:          EL $\leftarrow$ computeCrossEdges(C, $\mathcal{E}$, $\mathcal{E}_{prev}$, $rt_j$, itr)
22:      (mst, uf) $\leftarrow$ updateLocalMST(EL)
23:      $\mathcal{E}_{prev} \leftarrow \mathcal{E}$; $\mathcal{E} \leftarrow \mathcal{E}$ + CellSize *($2^{itr}$); itr++
       //end forall
24:    gmst $\leftarrow$ mergeMST(mst, p) //merge MSTs in tree parallel mode
     //end while
25: procedure mergeMST(mst, p)
26:    forall processor $P_i$, $1 \le i \le p$:
27:      for (k = 1 to $\log_2 p$)
28:        if ((i - 1)mod$2^k$ == 0)
29:          $mst_i \leftarrow$ mergeEdges($mst_i$ , $mst_j$) where $j=i+2^{k-1}$
       //end forall
30:    for process $P_1$
31:      (gmst, guf) $\leftarrow$ updateglobalMST(mst)
32:    broadcast guf to all $P_i$
33:    return gmst
```

After computing local MSTs, a synchronization step is performed to update global MST, *gmst*. For this, we maintain global MST and global *union-find* structures to get the final cluster membership of points. At master node, *EdgeLists* is updated by merging of local MSTs in tree parallel mode. For each edge in *EdgeLists* of master node if its end-points do not belong to the same cluster, we add that edge to *gmst*, and update the global *union-find* data structure accordingly. This completes one iteration of clustering i.e. construction of dendrogram up to a distance of $\mathcal{E}$ is completed. The global *union-find* (*guf*) is broadcast to all the nodes to remove redundant edges from *EL* in parallel. Moreover, the updated information is also used to optimize cross-edge computation in further iteration. Since the size of *guf* is *n* and *itr* is a small constant (max 12 across all experiments). Thus, the cost of sequential steps 30-32 would be O(n * $b(p)$), where $b(p)$ is the cost of broadcast time which is dependent on p and linearly increases with p. This will not affect the time complexity of the algorithm.

Steps 2 and 3 are repeated until the number of edges in global MST reach n-1.

*dGrid*SLINK preserves the correctness of *Grid*SLINK because both perform the same set of edge computations.

Complexity Analysis:

We derive the runtime complexity of *dGrid*SLINK in Table II. *dGrid*SLINK has a time complexity of $O\left(\frac{n^2}{p}\right)$ for p nodes i.e. the calculated speedup of *dGrid*SLINK is linear in the number of nodes.

TABLE II. TIME COMPLEXITY OF *dGrid*SLINK

S. No.	Steps	Time complexity
1.	Data distribution	$O(n * p)$
2.	Step 1-4 of *Grid*SLINK	$O\left(\frac{n^2}{p}\right)$
3.	Gather MST	$O(log\ RegionSize * n * p)$
4.	Global MST update	Same as *Grid*SLINK
5.	Total	$O\left(\frac{n^2}{p}\right)$

Iter is not affected by parallelization and thus in *dGrid*SLINK, its value will be same as that of *Grid*SLINK.

Data distribution step of *dGrid*SLINK is sequential. Traditionally, results reported in the literature do not include this step in their analysis. However, our experimental results include time taken by this step as well and it is a very small fraction of the total time (see Section V).

V. EXPERIMENTAL RESULTS

Distributed memory experiments are performed on a 32 nodes Beowulf cluster. Each node having single quad core processors (3.30GHz Intel(R) Xeon(R) 1230) and 32 GB RAM. There is a separate master node for cluster management and it is not used for computation. The nodes are connected by a 48-port, Gigabit Ethernet switch. The code is written in C using OpenMPI for parallel implementations to exploit distributed memory systems.

TABLE III. DATASETS SPECIFICATIONS

Dataset	*n*	*d*	*iter*
MPAGD3M3d	3,212,724	3	8
MPAGD16M3d	15,966,719	3	8
MPAGB3M3d	3,369,209	3	9
MPAGB8M3d	7,984,176	3	9
DGB8M3d	8,651,759	3	12
FOF380K6d	380,026	6	4
FOF4M6d	3,803,069	6	6
FOF5M10d	5,699,592	10	7
FOF57M3d	56,990,190	3	4
DGB0.5M3d	501,764	3	11
3DRN	434,874	3	6
MPAGD0.5M3d	559,219	3	6

For performance evaluation, we have considered various real datasets commonly used in literature for evaluating SLINK of varying sizes (434K to 57M) and dimensions (3-11): MPAGalaxiesDelucia2006a (MPAGD) [28], friend-of-friends (FOF) [28], MPAGalaxiesBertone2007a (MPAGB) [29], DGalaxiesBower2006a (DGB) [30], and DGalaxiesFont2008a (DGF) [31]. These datasets are obtained from the Galaxy and Halo databases on Millennium Run [31]. Details of all datasets used are given in Table III along with *iter*.

All time measurements are taken using VampirTrace version 5.14.2 [32]. The runtime measurement includes time for reading/writing the data from/to the disk and the time taken for distributing data to multiple threads/nodes. *R*-tree parameters m = 8 (min-entries) and M = 16 (max-entries) have been used for all experiments. For *d*Grid*SLINK*, we observed that the data distribution time is negligible as compared to local computation time. The time taken for data distribution is small – maximum 2.23% for FOF57M3d dataset and minimum 0.14% for FOF5M10d dataset of total computation time when using 32 nodes – for all datasets considered. Our algorithm shows considerable speedup either way.

The parameter, τ, represents density threshold whose value is found to be optimal in the range 300 to 3000 (see Fig. 8). In all experiments, we have taken τ minimum (τ = 300) to get small $CellSize_{init}$ thus achieving better load balancing. For cross-edge computation, remote data has to be communicated across the nodes. The amount of data, needs to be communicated, is dependent on $\mathcal{E}$-extended region, which is very small in initial iterations and maximum merging (approximately 80% to 90% [22]) happens in these iterations. Therefore, we give $\mathcal{E}$-extended data for few initial iterations at the beginning of iterations. This will reduce data communication. Moreover, if one seeks to have multiple clusters as a result of SLINK instead of single dendrogram, then no data communication is required. We have initially communicated remote data for four iterations in all our experiments.

The speedup results for *dGrid*SLINK is computed with respect to *Grid*SLINK. We have also benchmarked our proposed parallel algorithm against the best known distributed SLINK, PINK [12]. Our algorithm runs an order of magnitude faster than PINK and are capable of efficiently clustering datasets of sizes up to 57M (maximum size of datasets considered in literature is only 5M). For instance, *dGrid*SLINK takes 778.79 sec. to process FOF57M3d data. That is millions of data points can be clustered efficiently using less number of resources (see discussion of Fig. 11).

The performance of *Grid*SLINK depends on the number of points per cell (τ).The optimal value of τ tends to depend on the data size and has been determined experimentally. For the rest of the discussion, we show results with an appropriately chosen, not necessarily optimal, value of τ. Fig. 8 shows the variation in performance of *Grid*SLINK for different values of τ. In this particular case, the optimal value of τ is 300 for FOF380K6d and MPAGD0.5M3d datasets and 500 for DGB0.5M3d dataset although it is observed that a value of τ between 300 and 1000 yields near-optimal performance. For all datasets used in experiments, a value of τ between 300 and 3000 yields near-optimal performance.

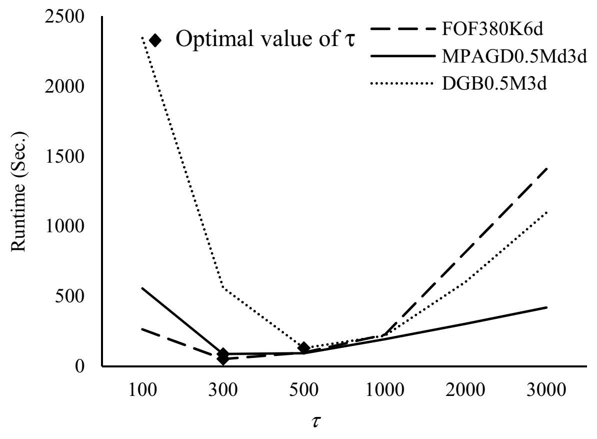

Fig. 8. Runtime of *Grid*SLINK for various datasets with increasing τ

The performance of *Grid*SLINK is compared with the sequential version of PINK (p=1) and it is found to consistently outperform PINK. We have used datasets of size up to 2.23M for comparison; because for larger size datasets, sequential PINK is found to be very slow (see Fig. 9). For instance, sequential PINK took 5.66 hours to cluster the MPAGD0.5M3d dataset. Table IV shows runtime ratio of sequential PINK (p=1) and *Grid*SLINK and the maximum speedup achieved is 233.58 for MPAGD0.5M3d.

TABLE IV. RUNTIME COMPARISON OF SEQUENTIAL VERSIONS FOR VARIOUS DATASETS (τ=300)

Dataset	PINK (p=1)	*Grid*SLINK	PINK/*Grid*SLINK Ratio
3DRN	11335.85	54.56	207.78
MPAGD0.5M3d	20362.83	87.18	233.58
DGB0.5M3d	16321.08	561.27	29.08

Fig. 9 highlights the scalability of *Grid*SLINK and comparison with PINK for increasing data. We have used MPAGD 3-dimensional dataset of varying size (varying in power of 2) up to 18.59M and τ=300. It is clear from the figure that the increase in runtime of *Grid*SLINK is far lower than that of PINK. We couldn't perform PINK execution beyond 2.23 million as it takes very long to execute.

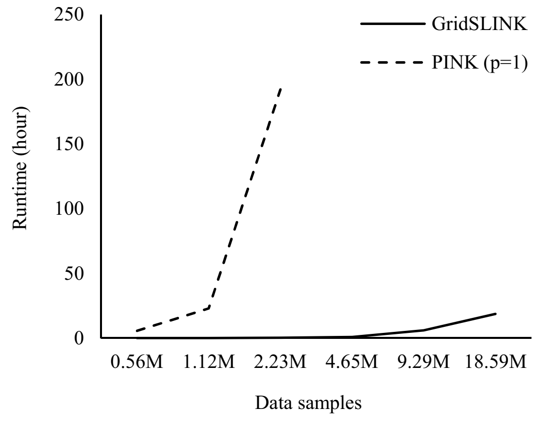

Fig. 9. Runtime of *Grid*SLINK and PINK for increasing data (τ=300)

The number of distance computations in *dGrid*SLINK and *Grid*SLINK are same therefore the ideal and theoretical (effective) speedups are same. Fig. 10 shows the speedup of *dGrid*SLINK on various datasets for increasing p. It can be observed from the figure that *dGrid*SLINK scales well for all datasets. The best speedup achieved is 21.44 for DGB8M3d dataset on 32 nodes. For p = 32, the optimal split was along one dimension for all datasets. However, we evaluated the performance for $l = 2$ ($k_1 = 16$ and $k_2 = 2$), and the speedup results are similar i.e. performance is not very sensitive to small changes in l, k_1, k_2, etc.

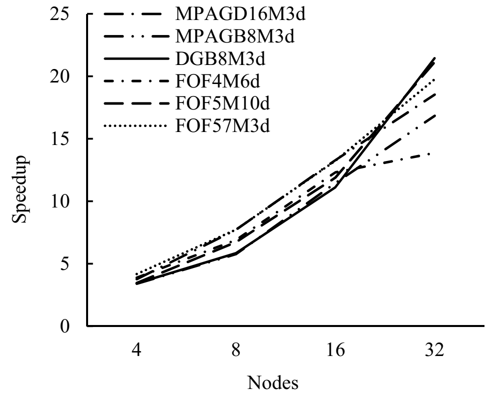

Fig. 10. Speedup of *dGrid*SLINK for increasing p

Fig. 11 shows the runtime ratio of *dGrid*SLINK with PINK on various datasets for 18 nodes. *dGrid*SLINK is up to 208.62x faster than PINK. *dGrid*SLINK and PINK complete their execution in 173.094 sec. and 36110.621 sec. respectively using 18 nodes for MPAGD3M3d. Then, PINK requires at least 3756 nodes to complete its execution in 173.094 sec. Therefore, we claim that the proposed parallel algorithm requires less number of resources than any existing parallel algorithm.

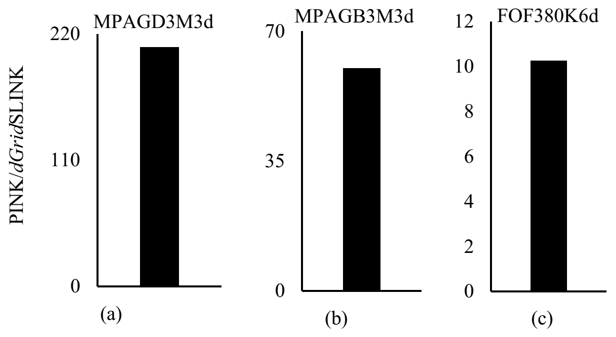

Fig. 11. Ratio of *dGrid*SLINK with PINK for p=18

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we present an efficient implementation of SLINK, *Grid*SLINK which runs an order of magnitude faster than the best existing sequential implementation of SLINK due to reduction in number of distance computations and efficient execution of neighborhood queries. A scalable distributed implementation of *Grid*SLINK, *dGrid*SLINK for commodity cluster is also presented. *dGrid*SLINK runs an order of magnitude faster (up to 208) than the best existing distributed implementation of SLINK due to efficient parallelization and optimized local computation through *Grid*SLINK. *dGrid*SLINK shows good speedup and scalability. The best speedup obtained is 21.44 for *dGrid*SLINK using 32 nodes. *dGrid*SLINK can execute on real datasets having millions of data points (up to 57M) in much less amount of time as compared to the best existing parallel algorithm. Alternatively, our algorithm requires fewer resources (nodes) to cluster large data while ensuring that the resulting dendrogram is the same as that produced by (classical) SLINK.

The proposed algorithms can further be optimized for cross-edge computations and data communications. This can be done by estimating a bound for all data points while computing *cross-edges* during initial iterations. Spatial locality can be further exploited to compute the bound.

Acknowledgment

This work was supported by a Research Grant from Department of Electronics and Information Technology (DeitY), Government of India.

References

[1] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[2] R. Sibson, “SLINK: An optimally efficient algorithm for the single-link cluster method,” *Comput. J.*, vol. 16, no. 1, pp. 30–34, Jan. 1973.

[3] F. Murtágh, *Multidimensional clustering algorithms*. Physica-Verlag, 1985.

[4] A. C. Wei-keng Liao Ying Liu, “A Grid-based Clustering Algorithm using Adaptive Mesh Refinement,” in *7th Workshop on Mining Scientific and Engineering Datasets of SIAM International Conference on Data Mining*, 2004.

[5] S. Goil, H. Nagesh, and A. Choudhary, “MAFIA: Efficient and scalable subspace clustering for very large data sets,” in *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1999, pp. 443–452.

[6] J. B. Kruskal, “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem,” *Proc. Am. Math. Soc.*, vol. 7, no. 1, pp. 48–50, Feb. 1956.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[8] “Parallel Algorithms for Hierarchical Clustering and Cluster Validity,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 11, pp. 1088–1092, Nov. 1990.

[9] C. Wu, S. Horng, and H. Tsai, “Efficient Parallel Algorithms for Hierarchical Clustering on Arrays with Reconfigurable Optical Buses,” *J. Parallel Distrib. Comput.*, vol. 60, no. 9, pp. 1137–1153, 2000.

[10] Z. Du and F. Lin, “A novel parallelization approach for hierarchical clustering,” *Parallel Comput.*, vol. 31, no. 5, pp. 523–527, May 2005.

[11] W. Hendrix, M. A. Patwary, A. Agrawal, W. Liao, and A. Choudhary, “Parallel Hierarchical Clustering on Shared Memory Platforms,” in *High Performance Computing (HiPC), 2012 19th International Conference on*, 2012, pp. 1–9.

[12] W. Hendrix and D. Palsetia, “A Scalable Algorithm for Single-Linkage Hierarchical Clustering on Distributed-Memory Architectures,” in *LDAV*, 2013, pp. 7–13.

[13] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, “A Scalable Hierarchical Clustering Algorithm Using Spark,” in *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, 2015, pp. 418–426.

[14] C. Jin, M. A. Patwary, A. Agrawal, W. Hendrix, W. Liao, and A. Choudhary, “DiSC : A Distributed Single-Linkage Hierarchical Clustering Algorithm using MapReduce,” in *Proceedings of the 4th International SC Workshop on Data Intensive Computing in the Clouds*, 2013.

[15] M. P. Forum, “MPI: A Message-Passing Interface Standard,” University of Tennessee, Knoxville, TN, USA, 1994.

[16] C. F. Olson, “Parallel Algorithms for Hierarchical Clustering,” *Parallel Comput.*, vol. 21, pp. 1313–1325, 1995.

[17] J. L. Bentley, “A parallel algorithm for constructing minimum spanning trees,” *J. Algorithms*, vol. 1, no. 1, pp. 51–59, 1980.

[18] V. Olman, F. Mao, H. Wu, and Y. Xu, “Parallel clustering algorithm for large data sets with applications in bioinformatics.,” *IEEE/ACM Trans. Comput. Biol. Bioinform.*, vol. 6, no. 2, pp. 344–352, 2009.

[19] C. Jin, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, “Incremental, Distributed Single-linkage Hierarchical Clustering Algorithm Using Mapreduce,” in *Proceedings of the Symposium on High Performance Computing*, 2015, pp. 83–92.

[20] E. Johnson and H. Kargupta, “Collective, Hierarchical Clustering from Distributed, Heterogeneous Data,” in *Large-Scale Parallel Data Mining*, vol. 1759, M. Zaki and C.-T. Ho, Eds. Springer Berlin Heidelberg, 2000, pp. 221–244.

[21] W. Hendrix, M. M. Ali Patwary, A. Agrawal, W.-K. Liao, and A. Choudhary, “Parallel Data Clustering Algorithms: Parallel Hierarchical Clustering Code.” 2012.

[22] M. Dash, S. Petrutiu, and P. Scheuermann, “pPOP: Fast yet accurate parallel hierarchical clustering using partitioning,” *Data Knowl. Eng.*, vol. 61, no. 3, pp. 563–578, Jun. 2007.

[23] S. Rajasekaran, “Efficient parallel hierarchical clustering algorithms,” *Parallel Distrib. Syst. IEEE Trans.*, vol. 16, no. 6, pp. 497–502, 2005.

[24] E. Dahlhaus, “Parallel Algorithms for Hierarchical Clustering and Applications to Split Decomposition and Parity Graph Recognition,” *J. Algorithms*, vol. 36, no. 2, pp. 205–240, Aug. 2000.

[25] M. Dash, H. Liu, P. Scheuermann, and K. L. Tan, “Fast hierarchical clustering and its validation,” *Data Knowl. Eng.*, vol. 44, no. 1, pp. 109–138, 2003.

[26] C. Faloutsos, T. Sellis, and N. Roussopoulos, “Analysis of Object Oriented Spatial Access Methods,” in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987, pp. 426–439.

[27] R. C. Prim, “Shortest Connection Networks And Some Generalizations,” *Bell Syst. Tech. J.*, vol. 36, no. 6, pp. 1389–1401, 1957.

[28] G. De Lucia and J. Blaizot, “The hierarchical formation of the brightest cluster galaxies,” *Mon. Not. Roy. Astron. Soc.*, vol. 375, pp. 2–14, 2007.

[29] S. Bertone, G. De Lucia, and P. A. Thomas, “The recycling of gas and metals in galaxy formation: Predictions of a dynamical feedback model,” *Mon. Not. Roy. Astron. Soc.*, vol. 379, pp. 1143–1154, 2007.

[30] R. G. Bower, A. J. Benson, R. Malbon, J. C. Helly, C. S. Frenk, C. M. Baugh, S. Cole, and C. G. Lacey, “Breaking the hierarchy of galaxy formation,” *Mon. Not. R. Astron. Soc.*, vol. 370, no. 2, pp. 645–655, Aug. 2006.

[31] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, “Simulations of the formation, evolution and clustering of galaxies and quasars,” *Nature*, vol. 435, no. 7042, pp. 629–636, Jun. 2005.

[32] H. Brunst, D. Hackenberg, G. Juckeland, and H. Rohling, “Comprehensive Performance Tracking with Vampir 7,” in *Tools for High Performance Computing 2009*, Springer Berlin Heidelberg, 2009, pp. 17–29.