CSI7106-01
MULTICORE COMPUTING TOPICS
PROJECT

July 3, 2020

Hyunmo Sung
Student ID: 2020311568

Yonsei University
Department of Computer Science

Contents

1 Introduction

For simplicity, graphs can be assumed to have at least 2 vertices. If it has only a single vertex then it will be a single vertex with or without a self-loop. For any graphs with more than 2 vertices, two vertices can be chosen from a graph. If there are possible paths from one vertex to another vertex. One of the paths' cost will be less than or equal to others. APSP is an abbreviation of the All-Pairs Shortest Paths problem. APSP is a problem that computes every shortest path for any pairs of vertices. This paper is trying to solve APSP at Sparse graph. Sparse graph has a small amount of edges. Naively, it meant to be $O(E) = O(V)$.

2 Notation

Graph G is defined by $G = (V, E)$ which V is a set of vertices and $E \subseteq V \times V$. At E, the first element denotes a source and the second element denotes a destination. Therefore, $(1, 2) \in E$ denotes that there is an edge from vertex 1 to vertex 2. Weighted graphs have weights for corresponding edges. In this report, W used to denote weights. $W(e)$ denotes a weight for $e \in E$. There are many algorithms to solve APSP. Some algorithms solve APSP with SSSP. SSSP is a Single-Source Shortest Path problem that computes every shortest path from a single source to every vertex. APSP can be solved by solving SSSP for every vertex. There are two big domains to solve SSSP which are Dijkstra's algorithm and Bellman-Ford algorithm. There are advantages and disadvantages to both algorithms. Dijkstra's algorithm is usually faster than Bellman-Ford, but it can't solve with negative edges but Bellman-Ford can solve with negative edges. Dijkstra's algorithm takes $O(|V|log|V| + |E|)$ for SSSP[1]. Therefore, Dijkstra's algorithm takes $O(|V|^2log|V| + |V||E|)$ for APSP. On the other hand, the Bellman-Ford algorithm takes $O(|V||E|)$ for SSSP, $O(|V|^2|E|)$ for APSP. Naively, $O(|E|)$ is between $O|V|$ and $O(|V|^2)$. Therefore, from the algorithmic aspect, Dijkstra's algorithm is faster than the Bellman-Ford algorithm. There is some extension known as Johnson's algorithm which removes every negative edge with solving SSSP with the Bellman-Ford algorithm and uses Dijkstra's algorithm to solve APSP[2]. With Johnson's algorithm, APSP can be solved with $O(|V|^2log|V| + |V||E|)$ with or without negative edges. However, it doesn't have much parallelism in the algorithm. Therefore, this paper tries to solve the algorithm with the Floyd-Warshall algorithm. Which has more potential for the parallelism.

3 Algorithm

Algotihms like Dijkstra's algorithm or Bellman-Ford algorithm are so-called edge relaxing algorithm. It relaxes edges with min operation for each time. From Floyd-Warshall algorithm, every shortest path goes through $(1, 2, \cdots, l)$ can be found at the end of iteration $k = l$. The detail of correctness is passed to [3] where Floyd-Warshall is from.

Algorithm 1 Floyd-warshall algorithm

```
1: function InitDistance(Dist, V, E)
2:     Dist ← two dimensional array with size of |V| × |V|
3:     for i ← 1, ··· , |V| do
4:         for j ← 1, ··· , |V| do
5:             if (V[i], V[j]) ∈ E then
6:                 Dist[i, j] ← W((i, j))
7:             else
8:                 Dist[i, j] ← ∞
9:             end if
10:        end for
11:    end for
12: end function
13: function Floyd-warshall(V, E)
14:    InitDistance(Dist, V, E)
15:    for k ← 1, ··· , |V| do
16:        for i ← 1, ··· , |V| do
17:            for j ← 1, ··· , |V| do
18:                Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])
19:            end for
20:        end for
21:    end for
22:    Return Dist
23: end function
```

From the algorithm above, all iterations for i and j can be parallelized. Floyd-Warshall requires only a synchronization point at the end of outer most loop. Therefore, it can have more parallelization points. There is a previous research to reduce the communication cost of Floyd-warshall algorithm at [4] which known as Blocked Floyd-Warshall algorithm. [4] uses blocks to solve APSP. It divides result distance matrix to multiple blocks and computes over blocks. This approach makes much better parallelism than the normal Floyd-Warshall algorithm. This paper tries to use this algorithm well on a sparse matrix. There are plenty of empty spaces in the sparse graph. Computation over empty space doesn't have a meaning at all. Empty space of distance matrix is denoting an ∞. With relaxing formula of Floyd-Warshall algorithm, three formula below works for computation with empty spaces.
$Dist[i,j] = \min(Dist[i,j], Dist[i,k] + \infty) = \min(Dist[i,j], \infty) = Dist[i,j]$
$Dist[i,j] = \min(Dist[i,j], \infty + Dist[k,j]) = \min(Dist[i,j], \infty) = Dist[i,j]$.
$Dist[i,j] = \min(Dist[i,j], \infty + \infty) = \min(Dist[i,j], \infty) = Dist[i,j]$.
From the formula above, it shows computation over empty space can be omitted. First, this paper tries to make a big empty space via Nested Dissection and make an elimination tree. Nested Dissection is a graph partitioning techinique to make S, C_1, C_2 from one graph $G = (V, E)$[5-6]. Nested Dissection tries to fulfill the requirements below.
1. S, C_1, C_2 are distinct vertices. 2. $S \cup C_1 \cup C_2 = V$ 3. For all $e = (s,t) \in E$, s,t are both in a S, C_1, C_2 or at least one of s,t needs to be at S. 4. $|S| \lll |C1| \simeq |C2|$
It is so-called "Nested" Dissection because it can be done at C_1, C_2 recursively. From Cholesky factorization there is a so-called elimination tree. In a nested dissection, elimination tree can be constructed as follow. 1. Do a nested dissection. 2. Make S to root. 3. Make C_1 and C_2 to children of S. 4. Do the same process for C_1, C_2. Therefore, every non-leaf node will be a separator and every leaf node will be a sort of C. From the elimination tree, it needs to compute over only parents and descendants. Notice that every C_1 and C_2 are connected to only S after nested dissection. Therefore, nodes don't need to compute over their siblings at the elimination tree. There are two funciton in use to describe the Super Floyd-Warshall algorithm. Two new function Ancestor and Descendant means as follows.
1. $\text{Parent}(k) = \{X | (X,K) \in E_t, k \in K\}$. 2. $\text{Child}(k) = \{X | (K,X) \in E_t, k \in K\}$.
3. $\text{Ancestor}(k) = \{X | X = \text{Parent}(k) \text{ or } X = Ancestor(Parent(k))\}$
4. $\text{Descendant}(k) = \{X | X = \text{Child}(k) \text{ or } X = Descendant(Child(k))\}$
Which $T = (V_t, E_t)$ is an elimination tree. Notice that nodes in T can have multiple vertices. From the elimination tree, siblings can proceed with their algorithm in parallel if it doesn't share the same data point. It has only collision at Stage 3. Therefore, it has a big parallelism in it.

4 Implementation

Algorithm 2 Parallel Super Floyd-warshall algorithm

```
1: function SUPER FLOYD-WARSHALL(V, E)
2:     InitDistance(Dist, V, E)
3:     Make M as 2D array of mutex with size of |V| by |V|.
4:     Make an elimination tree T = (V_t, E_t) with the nested dissection.
5:     Make S_1, S_2, S_3 as a stack for a computation of each stages.
6:     Do BFS with T to push every vertex in to S_1 in the order of depth.
7:     while S_1 is not empty do
8:         Dep ← Depth(Top(S_1)).
9:         while Top(S_1) = Dep do                                  ▷ Stage 1
10:            Fork SupernodeRelax(Dist, Top(S_1), Stage1Flag, M)
11:            Push Top(S_1) to S_2
12:            Pop S_1
13:        end while
14:        Wait for every thread to join
15:        while S_2 is not empty do                                ▷ Stage 2
16:            Fork SupernodeRelax(Dist, Top(S_2), Stage2Flag, M)
17:            Push Top(S_2) to S_3
18:            Pop S_2
19:        end while
20:        Wait for every thread to join
21:        while S_3 is not empty do                                ▷ Stage 3
22:            Fork SupernodeRelax(Dist, Top(S_3), Stage3Flag, M)
23:            Pop S_3
24:        end while
25:        Wait for every thread to join
26:    end while
27:    return Dist
28: end function
```

This algorithm uses SuperNodes to access vertices with the bottom-up approach. It first adds every supernode to the stack with BFS search. It keeps popping out and pushing it to the second stage stack until a depth changes. It uses these stacks to proceed with stage 2 and 3. It guarantees that it will proceed one depth by other and synchronizes results between stages.
It uses a SupernodeRelax to proceed with a stage with the intermediary in S. It passes Where to write, What are intermediaries are, Which stage it is, Mutex array as arguments. From the arguments, it forks multiple threads. It makes threads to iterate for every ancestors and descendant. If it is stage 1, it just does Floyd-Warshall over the corresponding superNode. If it is stage 2, it calls several threads. Three for columns and another three for rows, each will update columns or rows that have the same rows or columns with the intermediary. It iterates with Up direction or Down direction. If the direction is Up, it iterates to the parent. If the direction is Down, it iterates to children. It's the same for stage 3, but it needs to iterate for two sets. Notice that it iterates over $\{\text{Ancestor}(k) \cup \text{Descendant}(k)\} \times \{\text{Ancestor}(k) \cup \text{Descendant}(k)\}$. Therefore, two iterator needs to denote $\{\text{Ancestor}(k) \cup \text{Descendant}(k)\}$. Flag $Order$ used for denoting it. If it iterates for the first set, it denoted to $First$, otherwise $Second$. First, it starts iterations with the first order. After that, it starts iteration again in working threads. Finally, each thread will fork again and each of them will relax edges. The last implementation is about Nested dissection. Paper doesn't explain the detail of it, but it seems to use METIS for it. There are two implementations which this project used. One is using BFS to compute depths and choose the so-called most ideal case. $\text{num}(x)$ is used to denote how many vertices are in the depth x. For more detail, there is one function to choose the ideal case. Function ideality $Ideal(|S|, |C_1|, |C_2|) = \frac{|S|}{|C_1|+|C_2|} * (\frac{|C_1|}{|C_2|} + \frac{|C_2|}{|C_1|})$. It means good when it's small. This $Ideal(|S|, |C_1|, |C_2|)$ is used to fulfill the formula $|S| \lll |C_1| \simeq |C_2|$. With this function, it first chooses every vertex with fewest outgoing edges. After that, it does BFS to compute depth. It computes every possible cut by choosing some range of depths to the separator. It chooses the most ideal case which has the smallest ideality. However, the BFS approach has many problems to implement Nested Dissection. It can't determine the depth of none reachable parts since it uses BFS. In the project, it assumed to have the same depth for none reachable parts. Therefore, it shows a bad result in Nested dissection proceed in many cases like Tree and 1138 bus matrix. Therefore, METIS implementation tried either. However, METIS doesn't return separators. Paper doesn't explain much detail for it but it mentions Symbolic analysis at Sparse Direct Methods. However, it usually used for only Cholesky decomposition. Therefore, this project used an alternative method as BFS did. First, reorder matrix with METIS result. Second, Check empty spaces of the matrix. Third, try to compute ideality like BFS does and choose the best. As a result, it usually showed a nice Nested Dissection for every matrix.

5 Experiment

Graph	1138 bus	airfoil	bcswpwr09	delaunay n13	EB
Speed up(X)	1.27	7.8	1.70	13.49	1.69
Graph	jagmesh8	Linear	Minnesota	mplate	Planer
Speed up(X)	2.11	5.41	4.26	4.13	4.72
Graph	Tree	TSOPF RS b162 c1	uk	UsPowerGrid	netScience
Speed up(X)	1.56	14.49	6.63	6.57	1.22

Experiment has been done on the dual-socket E5-2697 CPU. Which has 56 threads in total. It already has much Speedup from classic Floyd-Warshall algorithm.

Graph	Linear graph
Classic Floyd-Warshall execution time(usec)	394855757
Blocked Floyd-Warshall execution time(usec)	23114474
Super Floyd-Warshall execution time(usec)	4595818
Blocked Floyd-Warshall Speed up(X)	17.08
Super Floyd-Warshall Speed up(X)	85.92

Execution time in each stage is like follow.

Graph	1138_bus	netScience	airfoil	TSOPF_RS_b162_c1
Post(%)	8.55	14.06	26.45	34.33
Stage 1(%)	2.01	0.79	0.84	1.54
Stage 2(%)	13.10	9.73	0.59	6.88
Stage 3(%)	75.14	75.91	65.98	57.03
Speedup (X)	1.27	1.22	7.8	14.49

It shows that it has more speed-up only when it took many time from post process like Nested dissection. Which means that it can be avoided to do Super Floyd-warshall algorithm by relatively short time of trial to analysis over Eliminiation tree. From elimination tree, exact size of thread call and load imbalance can be known ahead of computation.

6 Future works

There are some extensions possible more for the super Floyd-Warshall algorithm. One is doing a blocked Floyd-Warshall algorithm for SupernodeRelax() which will make better thread call for it. However, it needs more complex implementation and be careful to choose the atomic function or not. Another extension is about GPU implementation. If the block size is big enough, it can get more speed up with using GPU. The other extension is using OpenMP to implement parallelization. This paper can be guessed using OpenMP. Every extension is complex, but worthwhile.

References

1. Fredman, Michael L., and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." Journal of the ACM (JACM) 34.3 (1987): 596-615.
2. Johnson, Donald B. "Efficient algorithms for shortest paths in sparse networks." Journal of the ACM (JACM) 24.1 (1977): 1-13.
3. Warshall, Stephen. "A theorem on boolean matrices." Journal of the ACM (JACM) 9.1 (1962): 11-12.
4. Venkataraman, Gayathri, Sartaj Sahni, and Srabani Mukhopadhyaya. "A blocked all-pairs shortest-paths algorithm." Journal of Experimental Algorithmics (JEA) 8 (2003): 2-2. 5. George, Alan. "Nested dissection of a regular finite element mesh." SIAM Journal on Numerical Analysis 10.2 (1973): 345-363. 6. Lipton, Richard J., Donald J. Rose, and Robert Endre Tarjan. "Generalized nested dissection." SIAM journal on numerical analysis 16.2 (1979): 346-358.

Appendix 1. algorithms

Algorithm 3 Parallel Super Floyd-warshall algorithm

```
1: function SupernodeRelax(Dist, S, Flag, M)
2:     if Flag is Stage1Flag then
3:         for k ∈ S do
4:             for i ∈ S do
5:                 for j ∈ S do
6:                     Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])
7:                 end for
8:             end for
9:         end for
10:    else if Flag is Stage2Flag then
11:        Fork UpdateEdgeStage2(Dist, S, S's left ,Col, Down)
12:        Fork UpdateEdgeStage2(Dist, S, S's left ,Row, Down)
13:        Fork UpdateEdgeStage2(Dist, S, S's right ,Col, Down)
14:        Fork UpdateEdgeStage2(Dist, S, S's right ,Row, Down)
15:        Fork UpdateEdgeStage2(Dist, S, S's parent, Col, Up)
16:        UpdateEdgeStage2(Dist, S, S's parent, Row, Up)
17:        Wait for every thread to join
18:    else                                              ▷ Flag is Stage3Flag
19:        Fork UpdateEdgeStage3(Dist, S, S's left, S's left, M, First, Down, Down)
20:        Fork UpdateEdgeStage3(Dist, S, S's left, S's right, M, First, Down, Down)
21:        Fork UpdateEdgeStage3(Dist, S, S's left, S's parent, M, First, Down, Up)
22:        Fork UpdateEdgeStage3(Dist, S, S's right, S's left, M, First, Down, Down)
23:        Fork UpdateEdgeStage3(Dist, S, S's right, S's right, M, First, Down, Down)
24:        Fork UpdateEdgeStage3(Dist, S, S's right, S's parent, M, First, Down, Up)
25:        Fork UpdateEdgeStage3(Dist, S, S's parent, S's left, M, First, Up, Down)
26:        Fork UpdateEdgeStage3(Dist, S, S's parent, S's right, M, First, Up, Down)
27:        UpdateEdgeStage3(Dist, S, S's parent, S's parent, M, First, Up, Up)
28:        Wait for every thread to join
29:    end if
30: end function
```

Algorithm 4 Parallel Super Floyd-warshall algorithm

```
1: function UpdateEdgeStage2(Dist, S, T, Flag, Dir)
2:     if T = NULL then
3:         return
4:     end if
5:     if Dir = Up then
6:         Fork UpdateEdgeStage2(Dist, S, T's parent ,Flag, Up)
7:     else                                                    ▷ Dir = Down
8:         Fork UpdateEdgeStage2(Dist, S, T's left ,Flag, Down)
9:         Fork UpdateEdgeStage2(Dist, S, T's right ,Flag, Down)
10:    end if
11:    if Flag = Row then
12:        for k ∈ S do
13:            for i ∈ T do
14:                for j ∈ S do
15:                    Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])
16:                end for
17:            end for
18:        end for
19:    else                                                    ▷ Flag = Col
20:        for k ∈ S do
21:            for i ∈ S do
22:                for j ∈ T do
23:                    Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])
24:                end for
25:            end for
26:        end for
27:    end if
28:    Wait for every thread to join
29: end function
```

Algorithm 5 Parallel Super Floyd-warshall algorithm

```
1: function UpdateEdgeStage3(Dist, S, T1, T2, M, Order, Dir1, Dir2)
2:     if T1 = NULL or T2 = NULL then
3:         return
4:     end if
5:     if Dir1 = Up and Dir2 = Up then
6:         if Order = First then
7:             Fork UpdateEdgeStage3(Dist, S, T1's parent, T2, M, First, Up, Up)
8:         end if
9:         Fork UpdateEdgeStage3(Dist, S, T1, T2's parent, M, Second, Up, Up)
10:    else if Dir1 = Up and Dir2 = Down then
11:        if Order = First then
12:            Fork UpdateEdgeStage3(Dist, S, T1's parent, T2, M, First, Up, Down)
13:        end if
14:        Fork UpdateEdgeStage3(Dist, S, T1, T2's left, M, Second, Up, Down)
15:        Fork UpdateEdgeStage3(Dist, S, T1, T2's right, M, Second, Up, Down)
16:    else if Dir1 = Down and Dir2 = Up then
17:        if Order = First then
18:            Fork UpdateEdgeStage3(Dist, S, T1's left, T2, M, First, Down, Up)
19:            Fork UpdateEdgeStage3(Dist, S, T1's right, T2, M, First, Down, Up)
20:        end if
21:        Fork UpdateEdgeStage3(Dist, S, T1, T2's parent, M, Second, Down, Up)
22:    else                                          ▷ Dir1 = Down and Dir2 = Down
23:        if Order = First then
24:            Fork UpdateEdgeStage3(Dist, S, T1's left, T2, M, First, Down, Down)
25:            Fork UpdateEdgeStage3(Dist, S, T1's right, T2, M, First, Down, Down)
26:        end if
27:        Fork UpdateEdgeStage3(Dist, S, T1, T2's left, M, Second, Down, Down)
28:        Fork UpdateEdgeStage3(Dist, S, T1, T2's right, M, Second, Down, Down)
29:    end if
30:    for k ∈ S do
31:        for i ∈ T1 do
32:            for j ∈ T2 do
33:                if Dir1 = Up and Dir2 = Up then
34:                    if Dist[i, k] + Dist[k, j] < Dist[i, j] then
35:                        Lock a mutex M[i, j]
36:                        if Dist[i, k] + Dist[k, j] < Dist[i, j] then
37:                            Dist[i, j] = Dist[i, k] + Dist[k, j]
38:                        end if
39:                        Unlock a mutex M[i, j]
40:                    end if
41:                else
42:                    Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])
43:                end if
44:            end for
45:        end for
46:    end for
47:    Wait for every thread to join
48: end function
```

Algorithm 6 Nested dissection with BFS

function NesteDissectionCut(V, E, S)
  if $|V| <$ threshold **then**
    return S
  end if
  Count outgoing edges from every vertex v and denote it as $E^{out}(v)$
  $Best \leftarrow (\infty, 0, 0)$
  for v in $\{v | E^{out}(v) = \min(E^{out}(V))\}$ **do**
    Do BFS to compute depth.
    for $i \leftarrow \{1, \cdots, \max(depth) - 2\}$ **do**
      for $j \leftarrow \{i, \cdots, \max(depth) - 1\}$ **do**
        $CurrentIdeal \leftarrow Ideal(\Sigma_{k=j+1}^{\max(depth)} num(k), \Sigma_{k=1}^{i} num(k), \Sigma_{k=i+1}^{j} num(k))$
        if $Best[0] > CurrentIdeal$ **then**
          $Best \leftarrow (CurrentIdeal, i, j)$
        end if
      end for
    end for
  end for
  Make a supernode C_1, C_2
  for $v \in V$ **do**
    if $depth(v) \leq Best[1]$ **then**
      Push v to C_1
      Pop v from S
    else if $depth(v) \leq Best[2]$ **then**
      Push v to C_2
      Pop v from S
    end if
  end for
  C_1's parent $\leftarrow S$
  C_2's parent $\leftarrow S$
  S's left $\leftarrow C_1$
  S's right $\leftarrow C_2$
  $C_1 \leftarrow$ NesteDissectionCut(V, E, C_1)
  $C_2 \leftarrow$ NesteDissectionCut(V, E, C_2)
  return S
end function
function NestedDissection(V, E)
  Make a Super node S containing every vertex in it.
  $S \leftarrow$ NesteDissectionCut(V, E, S)
  return S
end function

Algorithm 7 Nested dissection with METIS

function NesteDissectionCut(V, E, S)
  if $|V| <$ threshold **then**
    return S
  end if
  $Best \leftarrow (\infty, 0, 0)$
  for $i \leftarrow \{1, \cdots, |V| - 2\}$ **do**
    for $j \leftarrow \{1, |V| - 1 - i\}$ **do**
      if All $E(V[i], V[j]) = \infty$ **then**
        $CurrentIdeal \leftarrow Ideal(|V| - i - j, i, j)$
        if $Best[0] > CurrentIdeal$ **then**
          $Best \leftarrow (CurrentIdeal, i, j)$
        end if
      end if
    end for
  end for
  Make a supernode C_1, C_2
  for $i \in \{1, \cdots, Best[1]\}$ **do**
    Push $V[i]$ to C_1
  end for
  for $j \in \{Best[1] + 1, \cdots, Best[1] + Best[2]\}$ **do**
    Push $V[j]$ to C_2
  end for
  for $i \in \{1, \cdots, Best[1] + Best[2]\}$ **do**
    Pop $V[0]$
  end for
  C_1's parent $\leftarrow S$
  C_2's parent $\leftarrow S$
  S's left $\leftarrow C_1$
  S's right $\leftarrow C_2$
  $C_1 \leftarrow$ NesteDissectionCut(V, E, C_1)
  $C_2 \leftarrow$ NesteDissectionCut(V, E, C_2)
  return S
end function
function NestedDissection(V, E)
  $Perm \leftarrow$ METIS_NodeND(V, E)
  Make a Super node S containing every vertex in it with order of $Perm$.
  $S \leftarrow$ NesteDissectionCut(V, E, S)
  return S
end function

Appendix 2. matrix after reordering

Red boxs denote supernodes, blue boxs denote corresponding seperators.

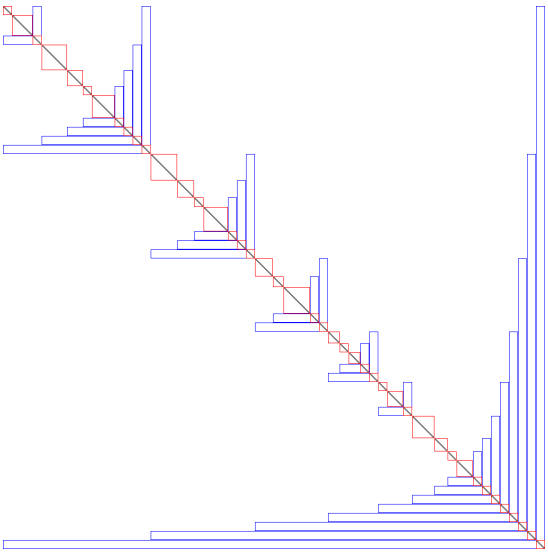

Figure 1: BFS Nested Dissection of Linear graph

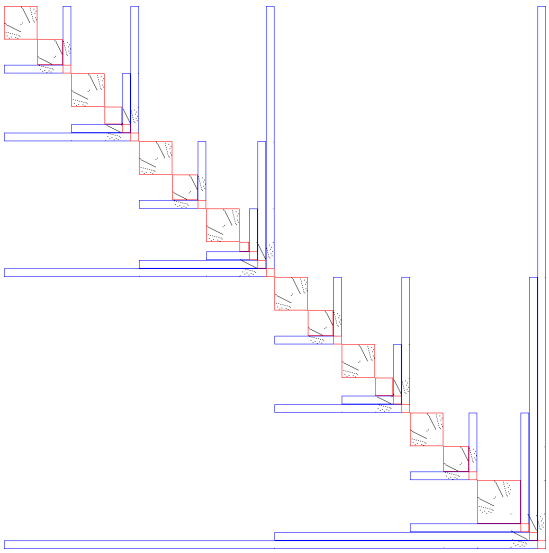

Figure 2: METIS Nested Dissection of Linear graph

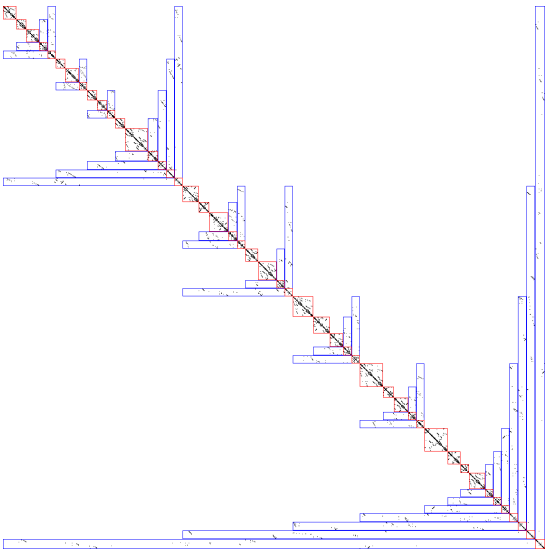

Figure 3: BFS Nested Dissection of Jagmesh

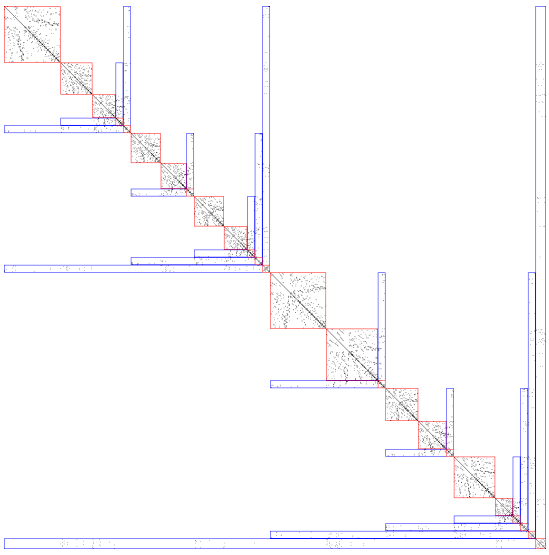

Figure 4: METIS Nested Dissection of Jagmesh

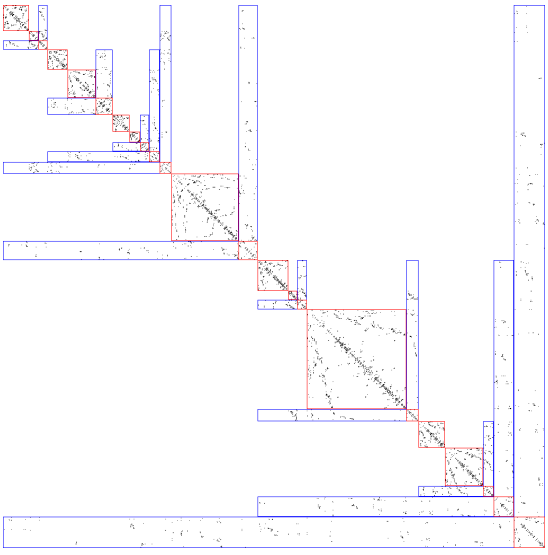

Figure 5: BFS Nested Dissection of delaunay

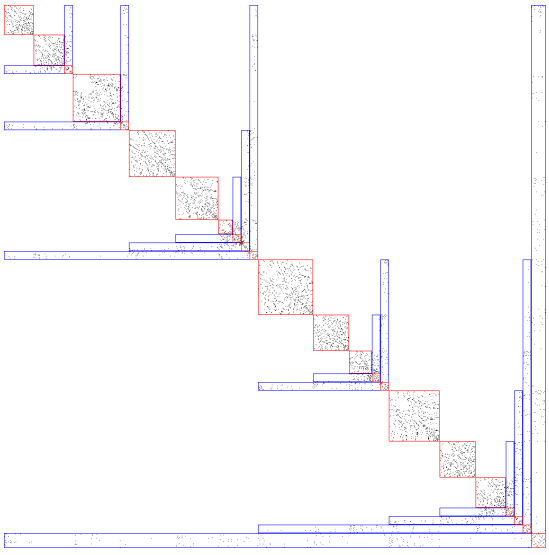

Figure 6: METIS Nested Dissection of delaunay

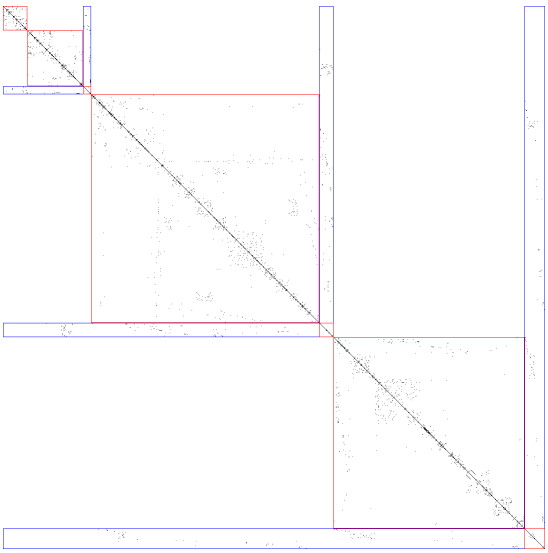

Figure 7: BFS Nested Dissection of bus graph

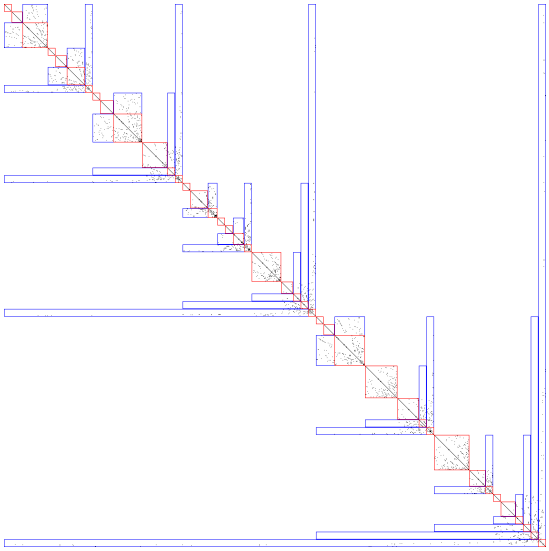

Figure 8: METIS Nested Dissection of bus graph

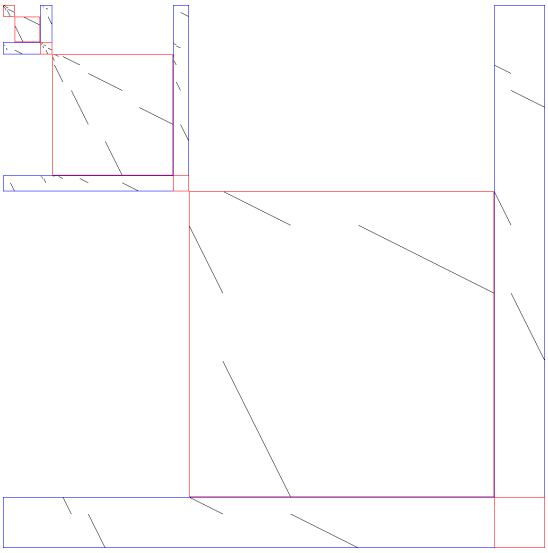

Figure 9: BFS Nested Dissection of Tree graph

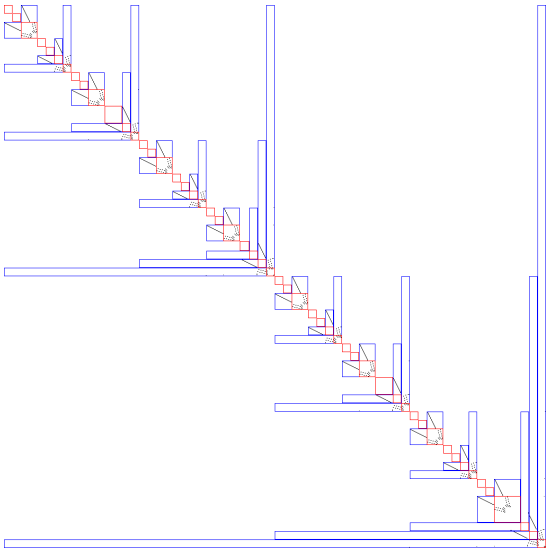

Figure 10: METIS Nested Dissection of Tree graph

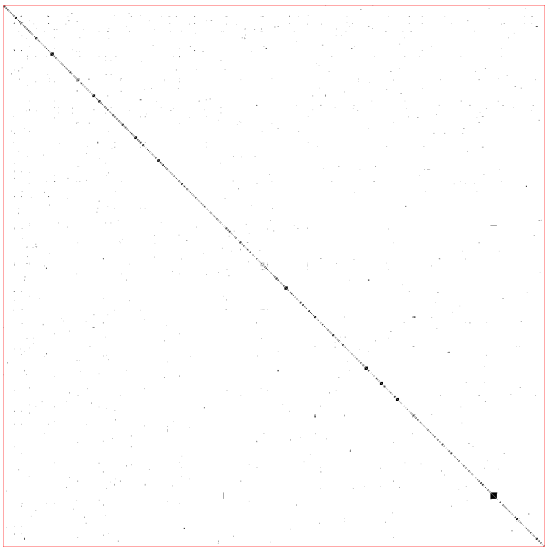

Figure 11: BFS Nested Dissection of Netscience

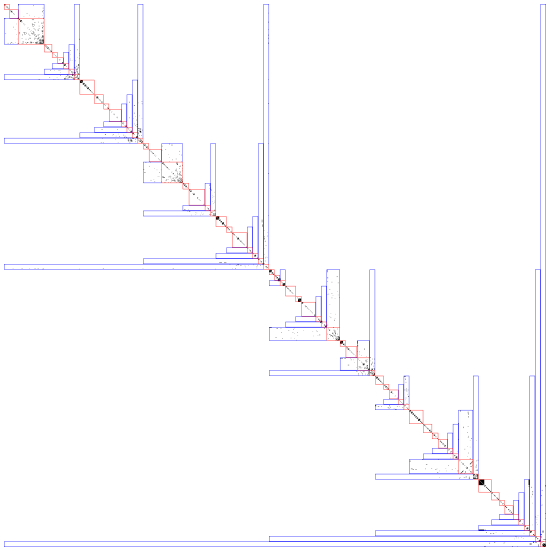

Figure 12: METIS Nested Dissection of Netscience