

Solution of Sparse Linear Systems on a Cluster of Workstations using Graph Partitioning Methods

Kayhan Erciyeş, Betül Turhal
Ege University International Computer Institute,
35100 İzmir, Turkey, Tel 0-232-339 9114
e-mail:erciyesh@ube.ege.edu.tr, turhal@bornova.ege.edu.tr

October, 2000

Abstract

We propose a method to solve large sparse linear systems directly on a parallel processing environment consisting of a cluster of workstations. The sparse coefficient matrix represented as a graph is first partitioned into n regions of approximately equal number of nodes using suitable heuristics. The sparse linear system is then solved using all-to-all communication.

Keywords: Sparse linear systems, graph partitioning, all-to-all communication.

1 Introduction

We propose a method to solve large sparse linear systems directly on a cluster of workstations. The system comprises the following:

1. Graph Partitioning Algorithm
2. Communication Protocol
3. Sparse Linear Solvers

The graph partitioning algorithm which uses original heuristics is explained in section 2 with the outline of the obtained results. The communication protocol which employs a logical ring on the workstations is given in section 3. The actual solvers which communicate the results using the ring protocol and future directions are described in section 4.

2. Graph Partitioning Methods

The graph partitioning problem is to partition the vertices in p roughly equal partitions, such that the number of edges connecting vertices in different partitions is minimized. This problem finds applications in many areas including parallel scientific computing, task scheduling, and VLSI design. Some examples are domain decomposition for minimum communication mapping in the parallel execution of sparse linear system solvers, mapping of spatially related data items in large geographical information systems on disk to minimize disk I/O requests, and mapping of task graphs to parallel processors.

For example, the solution of a sparse system of linear equations $Ax = b$ via iterative methods on a parallel computer gives rise to a graph partitioning problem. A key step in each iteration of these methods is the multiplication of a sparse matrix and a (dense) vector. A good partition of the graph corresponding to matrix A can significantly reduce the amount of communication in parallel sparse matrix-vector multiplication. If

parallel direct methods are used to solve a sparse system of equations, then a graph partitioning algorithm can be used to compute a fill reducing ordering that lead to high degree of concurrency in the factorization phase. The multiple minimum degree ordering used almost exclusively in serial direct methods is not suitable for parallel direct methods, as it provides very little concurrency in the parallel factorization phase.

A class of graph partitioning algorithms reduces the size of the graph (i.e., coarsen the graph) by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. These are called multilevel graph partitioning schemes. Some researchers investigated multilevel schemes primarily to decrease the partitioning time, at the cost of somewhat worse partition quality. Recently, a number of multilevel algorithms have been proposed that further refine the partition during the uncoarsening phase. These schemes tend to give good partitions at a reasonable cost. Random maximal matchings are used to successively coarsen the graph down to a few hundred vertices. Edge and vertex weights to capture the collapsing of the vertex and edges are also used.

2.1 Graph Partitioning

Partition a graph into k partitions problem is defined as follows: Given a graph $G = (V, E)$ with $|V| = n$, partition V into k subsets, V_1, V_2, \dots, V_k such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$, and $\cup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subsets is minimized. The graph partitioning problem can be naturally extended to graphs that have weights associated with the vertices and the edges of the graph. In this case, the goal is to partition the vertices into k disjoint subsets such that the sum of the vertex-weights in each subset is the same, and the sum of the edge-weights whose incident vertices belong to different subsets is minimized.

The efficient implementation of many parallel algorithms usually requires the solution to a graph partitioning problem, where vertices represent computational tasks, and edges represent data exchanges. Depending on the amount of the computation performed by each task, the vertices are assigned a proportional weight. Similarly, the edges are assigned weights that reflect the amount of data that needs to be exchanged. A partitioning of this computation graph can be used to assign tasks to k processors. Since the partitioning assigns to each processor tasks whose total weight is the same, the work is balanced among k processors. Furthermore, since the algorithm minimizes the edge-cut (subject to the balanced load requirements), the communication overhead is also minimized.

2.1.1 Multilevel Graph Bisection

The graph G can be bisected using a multilevel algorithm. The basic structure of a multilevel algorithm is very simple. The graph G is first coarsened down to a few hundred vertices, a bisection of this much smaller graph is computed, and then this partition is projected back towards the original graph (finer graph). At each step of the graph uncoarsening, the partition is further refined. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut. Formally, a multilevel graph

bisection algorithm works as follows: Consider a weighted graph $G_0=(V_0,E_0)$, with weights both on vertices and edges. A multilevel graph bisection algorithm consists of three phases. It begins with the coarsening phase. The graph G_0 is transformed into a sequence of smaller graphs G_1, G_2, \dots, G_m such that $|V_0| > |V_1| > |V_2| > \dots > |V_m|$. The second phase is the partitioning phase. A 2-way partition P_m of the graph $G_m=(V_m,E_m)$ is computed that partitions V_m into two parts, each containing half the vertices of G_0 . The uncoarsening phase will be the last phase of the multilevel graph bisection. The partition P_m of G_m is projected back to G_0 by going through intermediate partitions P_m

2.2 Coarsening Phase

During the coarsening phase, a sequence of smaller graphs, each with fewer vertices, is constructed. Graph coarsening can be achieved in various ways. Some possibilities are shown in Figure 2.1

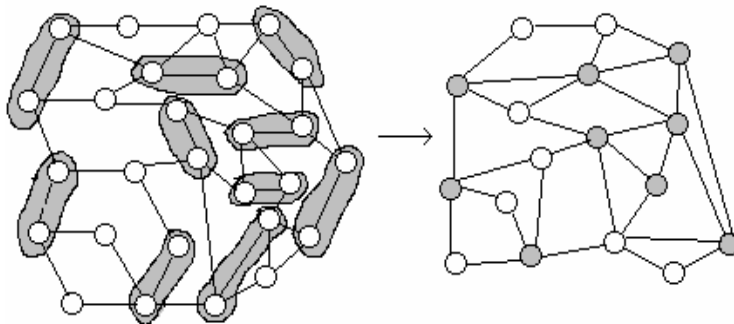


Figure 2.1 Coarsening operation.

In most coarsening schemes, a set of vertices of G_i is combined to form a single vertex of the next level coarser graph G_{i+1} . Let $V v_i$ be the set of vertices of G_i combined to form vertex v of G_{i+1} . We will refer to vertex v as a multinode. In order for a bisection of a coarser graph to be good with respect to the original graph, the weight of vertex v is set equal to the sum of the weights of the vertices in $V v_i$. Also, in order to preserve the connectivity information in the coarser graph, the edges of v are the union of the edges of the vertices in $V v_i$. In the case where more than one vertex of $V v_i$ contain edges to the same vertex u , the weight of the edge of v is equal to the sum of the weights of these edges. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening phase, the bisection is successively refined as it is projected to the larger graphs.

Two main approaches have been proposed for obtaining coarser graphs. The first approach is based on finding a random matching and collapsing the matched vertices into a multinode, while the second approach is based on creating multinodes that are made of groups of vertices that are highly connected. The later approach is suited for graphs arising in VLSI applications, since these graphs have highly connected components.

However, for graphs arising in finite element applications, most vertices have similar connectivity patterns (i.e., the degree of each vertex is fairly close to the average degree of the graph).

Here are some basic ideas behind coarsening using matchings. Given a graph $G_i = (V_i, E_i)$, a coarser graph can be obtained by collapsing adjacent vertices. Thus, the edge between two vertices is collapsed and a multinode consisting of these two vertices is created. This edge collapsing idea can be formally defined in terms of matchings. A matching of a graph, is a set of edges, no two of which are incident on the same vertex. Thus, the next level coarser graph G_{i+1} is constructed from G_i by finding a matching of G_i and collapsing the vertices being matched into multinodes. The unmatched vertices are simply copied over to G_{i+1} . Since the goal of collapsing vertices using matchings is to decrease the size of the graph G_i , the matching should contain a large number of edges. For this reason, maximal matchings are used to obtain the successively coarse graphs. A matching is maximal if any edge in the graph that is not in the matching has at least one of its endpoints matched.

2.2.1 Centered node Matching (CM)

This is the matching method we implement and suggest. We compare the HEM and OHEM methods (described below) with the CM. We get the best partitions using centered matching method. Using the BFS (breadth first search) algorithm we try to find out the most appropriate center nodes. Appropriate means nodes with the at least some predetermined distance to each other. On each graph number of center nodes is equal to number of aimed partitions of the graph. (# of workstations) and coarsen the graph with the given algorithm below until the graph has only the assigned center nodes.

1. $C_i \in$ Center node set. $i = 0 \dots \#$ of workstations
2. $i = 0$;
3. while (number of members in our adjacency list $>$ # of workstations) do
4. Find C_i in the adjacency list
5. Select the first neighbour node N of C_i in the adjacency list for the coarsening operation.
6. if $N \in$ Center node set then select the next neighbour of C_i as N .
7. if all N 's in the neighbour list \in Center node set then
8. $i=i+1$ and continue
9. Find N in the adjacency list
10. Append the neighbour list of node N to the neighbour list of C_i .
11. Add the edge set of node N to the edge set of C_i .

12. Add the set of b values of node N to the set of b values of Ci
13. Remove all N edges in the neighbour list of Ci
14. Link all nodes which are connected to node N to node Ci
15. Remove node N from the adjacency list.
16. $i = i+1 \text{ mod } (\# \text{ of workstations})$
17. endwhile /*Line 3*/
18. end /*BFS*/

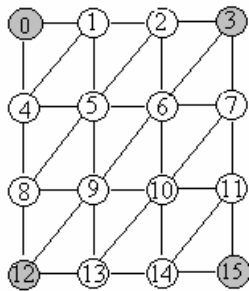


Figure 2.2 Selected center nodes in a regular graph

2.2.3 Heavy Edge and Ordered Heavy Edge Matching (HEM and OHEM)

Finding a maximal matching that contains edges with large weight is the idea behind the heavy-edge matching. A heavy-edge matching is computed using a randomized algorithm. The vertices are again visited in random order. However, instead of randomly matching a vertex u with one of its adjacent unmatched vertices, we match u with the vertex v such that the weight of the edge (u,v) is maximum over all valid incident edges (heavier edge). Our overall goal is to find a partition that minimizes the edge-cut. Consider a graph $G_i = (V_i, E_i)$, a matching M_i that is used to coarsen G_i , and its coarser graph $G_{i+1} = (V_{i+1}, E_{i+1})$ induced by M_i . If A is a set of edges, define $W(A)$ to be the sum of the weights of the edges in A . HEM contains only one edge in A . But OHEM finds a maximal matching on the graph. It can be shown that

$$W(E_{i+1}) = W(E_i) - W(M_i).$$

Thus, the total edge-weight of the coarser graph is reduced by the weight of the matching. Hence, by selecting a maximal matching M_i whose edges have a large weight, we can decrease the edge-weight of the coarser graph by a greater amount. As the analysis in shows, since the coarser graph has smaller edge-weight, it also has a smaller edge-cut. Note that this algorithm does not guarantee that the matching obtained has maximum weight (over all possible matchings), but our experiments have shown that it works very well. The complexity of computing a heavy-edge matching is $O(|E|)$, which is asymptotically similar to that for computing the random matching.

2.3 Uncoarsening Phase

During the uncoarsening phase, the partition P_m of the coarser graph G_m is projected back to the original graph, by going through the graphs $G_{m-1}, G_{m-2}, \dots, G_1$. Since each vertex of G_{i+1} contains a distinct subset of vertices of G_i , obtaining P_i from P_{i+1} is done by simply assigning the set of vertices V_i^v collapsed to $v \in G_{i+1}$ to the partition $P_{i+1}[v]$ (i.e., $P_i[u] = P[v], \forall u \in V_i^v$). Even though P_{i+1} is a local minimum partition of G_{i+1} , the projected partition P_i may not be at a local minimum with respect to G_i . Since G_i is finer, it has more degrees of freedom that can be used to improve P_i , and decrease the edge-cut.

3. PROTOCOL

Our protocol consist of two combined architectures. There are n servers and one client. The graph partitioning is done by the client which communicates in a star architecture with the workstations (servers). The elimination and solving x values operations are done by the servers who are communicating in a ring architecture (Figure 2.1). The client process create one assigned thread for each server. Each server communicates only with its assigned thread in the client process. The threads in the client process are running parallel and independent from each other. The connection is established using the transmission control protocol (TCP) which is a connection-oriented protocol. TCP provides a reliable, full-duplex, byte stream for a user process.

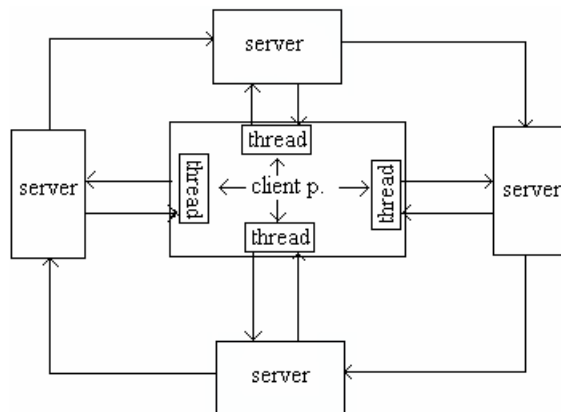


Figure 3.1 Protocol architecture

Each Workstation has 3 processes (threads) running in parallel. The server process, communication and solve-elimination thread. These processes are discussed in detail in this section. The client process embed our sparse linear system into a graph. The data of our sparse linear system becomes parallel using graph partition methods, as described before. An initial data is sent to each server which includes information about their position in the ring architecture and the amount of data they have to expect from the thread they are communicating. After all data is sent to each server, the servers begin to solve the system and communicate with their neighbors in the ring architecture.

3.1 Client Process

Graph partitioning and sending parallelized data to servers is done by the client process. The client process creates n threads (n is the number of servers). Each i th thread establish a connection with the i th server which are communicating in a in the ring architecture. Initial data is sent from i th thread and the i th server which includes information about the next communication parameters for the i th server. The initial data includes information like the edge number, b values number the i th server has to expect, its place in the ring architecture, who its neighbor in the ring architecture is and which socket numbers have to be used for the communication with its neighbour. After sending initial data to each server each thread send parallelized data to the server it communicating with. The thread begins to wait for a message from the server it communicates that it enters the "forward mode". (Figure 3.2) The "forward mode" means that there is nothing more to solve on the server and that the server who enters the forward mode will only forward the incoming frames to its neighbor in the ring architecture. Each thread increase a global flag in the client process by one. If the value of the flag is equal to the number of servers each thread send a termination message to the server it communicates (Figure 3.3)

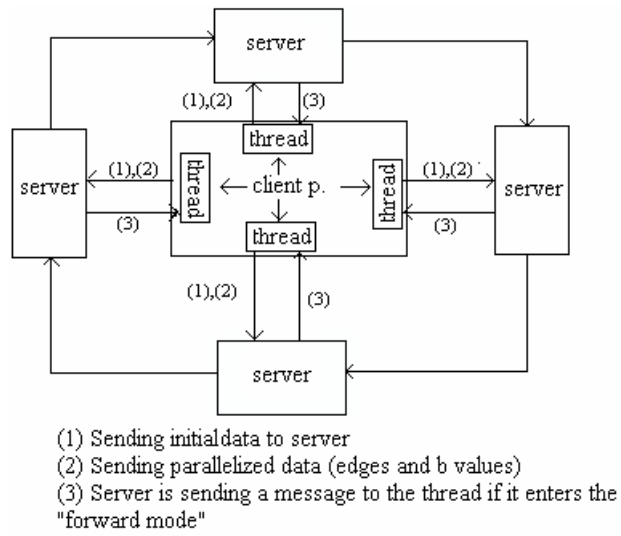


Figure 3.2 Protocol steps between client and server process

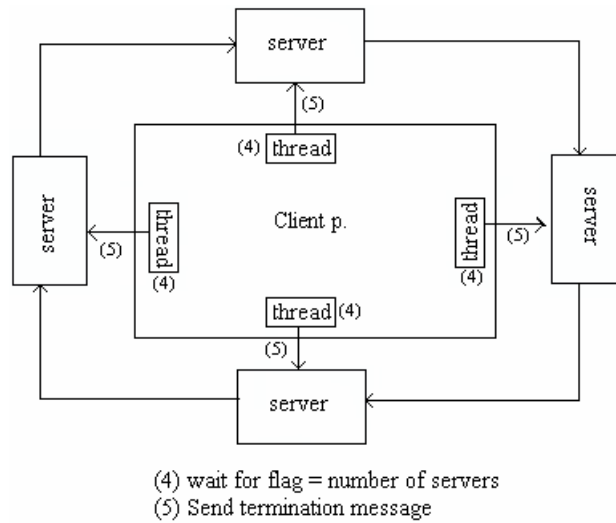


Figure 3.3 Completion of the operation

3.2 Server Process

The server process is the part of our application which tries to solve our linear system. It runs on distributed workstations and are communicating with each other within a ring architecture. We take our results using 2, 4 and 8 Workstations.

The server process begins by waiting for a connection request from one thread, which will be assigned to this server, in the client process. After connection establishment a message containing the initial data will be received. This message includes parameters for the continuing communication steps with the other server processes in the ring architecture. These parameters inform the server process about its process id in the ring architecture (its place in the ring), who its neighbors are (each server will only communicate with its neighbor in the ring architecture), how many edge values and b values it has to expect from the thread in the client process. Receiving this initial data the server process try to establish a connection to its neighbor workstation and after that it receives the edge and b values from the thread in the client process. Using the received data a subgraph of the original graph in the client process is reconstructed. We can also say that each server uncoarse its received coarsed part of our original graph.

If we summarize, we now have a connection to our neighbor server process, a connection to one assigned thread for us in the client process and a subgraph of our sparse linear system.

For parallelising the elimination-solving operation and the communication operation, a thread which do all communication operations is created (Communication thread). The next step is the "first elimination" step. Each server process apply the elimination procedure to eliminate its subgraph without any communication with its neighbors. One more thread is created when the "first elimination" is done for elimination and solving x values (elimination-solving thread).

3.2.1 Elimination-Solve Thread

The elimination-solve thread try to shrink the graph until the graph has no node anymore. It gets frames from the communication thread, convert to adjacency lists and use them as input for the elimination and solve procedures. The elimination-solve thread gets only frames from the communication thread.

Frames which find no node to eliminate in this server could be used in the future. A frame with the first col value fc (the smallest neighbor id of the sended node) need to find a node which first neighbor id fn_i is equal to fc . If we can't find such a fn_i this does not mean that there will never such a fn_i .

Lets say for example that $fc = 5$ but there are only $fn_i < 5$ and $fn_i > 5$ in our subgraph. Lets think of the next frame received from the communication thread and assume that due to that frame one node which $fn_i = 5$ is obtained in our new subgraph. So if we had received these two frames in the reverse order we could use both of them not only the second. To overcome this problem we store each frame in the "repeat elimination" set which supply this condition

$$\exists fn_i \text{ in the subgraph } fn_i < fc.$$

If the frame type of the received frame is 1 it places the x values to its x-value list which contains the solved x values and apply the solve procedure. If the frame type is 0, then it converts the frame to a adjacency list structure and apply the elimination procedure with this adjacency list as procedure input. After one of these two operations the repeat-elim set is checked. Every frame in this set try again to eliminate the subgraph.

These steps are repeated until no node remains in the graph. This means that there is nothing more to solve on this server and that the "forward mode" has to be entered. The elimination-solve thread sets the forward-flag and terminates.

Solve-Elimination Thread

```

1. while(1){
2.     flag = 0
3.     if elim-repeat set has any member then
4.         for all elim-repeat members begin
5.             Apply elimination procedure with the      elim-repeat member
               as input
6.             if elimination procedure output = 2 then set flag value (this means
               the elim-repeat member could find a node to eliminate in the graph.
7.             if elimination procedure output = 0 then remove the elim-repeat
               member from the elim-repeat set. ( This occurs when the first-col
               value of the elim-repeat member has a smaller value than any first-
               col value in the graph)
8.         endfor /*Line 4*/
9.     endif /*Line 3*/
10.    if flag = 1 repeat step 3
11.    apply solve procedure
12.    if the graph has no node anymore then
13.        set forward_flag
14.        terminate
15.    endif /*Line 12*/
16.    wait for frame from the communication thread
17.    convert frame to adjacency list structure
18.    if received frame type = 0 then
19.        apply elimination procedure with received frame as input.
           if elimination procedure output = 1 then
           add frame to the elim-repeat set. ( this means there are nodes in the graph
           which firstcol value is smaller than the first col value of the frame)
20.    endif /*Line 18*/
21.    if received frame type = 1 then
22.        add new x values to x values list
23.        apply solve procedure
24.    endif /*Line 21*/
25.    if the graph has no node anymore then
26.        set forward_flag
27.        terminate
28.    endif /*Line 25*/

```

3.2.2 Elimination Procedure

The aim of our elimination procedure is the same as in the original gauss-elimination. The original gauss elimination begins with the first row and try to make all first non-zero coefficients which are under our selected row and share the same column of the first non-zero coefficient in our selected row zero. And repeat this operation until the $n-1$. th row. If we return to our graph structure the first non-zero entries in same columns means the first neighbor of a node in the adjacency list structure. So we transfer the similar idea of the Gauss-elimination from a $n \times n$ matrix to a graph structure with one difference. The difference is that the selected node (row in $n \times n$ matrix) do not only search for suitable nodes under itself in the adjacency list structure it search in the whole graph and apply the known elimination steps except on itself. While applying the elimination steps, the edge between the first common neighbor and the node which supply this condition will be removed. The values of the remaining adjacency members will be updated. The same as in the original Gauss elimination. The first non-zero coefficient becomes zero (edge is removed) and the values of the remaining coefficients changes.

The elimination procedure is dealing with two main structures. An array of adjacency lists (our graph) and a second adjacency list say elim-list (Figure 3.4).

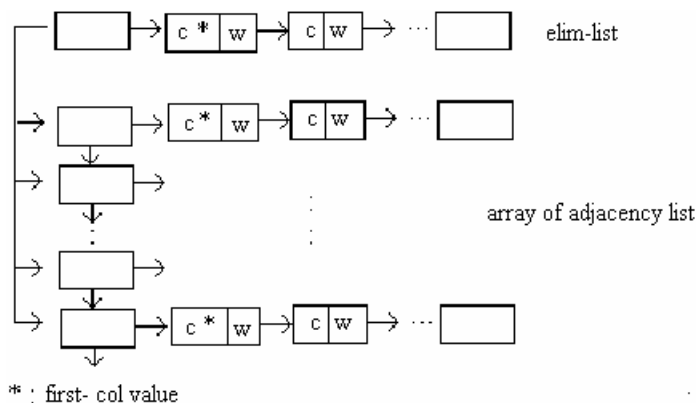


Figure 3.4 Elimination procedure components

The elim-list try to find a adjacency list in our graph with the same first-col value. The elim-list can only eliminate adjacency lists with the same first-col value. If the graph has adjacency lists which first col value is smaller than the first col value of the elim list the elim-flag is set to 1. This flag is used for the repeat-elimination set. If this elim-list is not a member of the repeat-elim set and is derived from an other server it is added to the repeat-elim set after completion of the elimination procedure. If the elim-list could not find a adjacency list to eliminate it terminates and return the elim-flag as output.

Otherwise if the elim-list find a suitable adjacency list it set the elim-flag to 2, enqueue the pointer value of this adjacency list to a queue of adjacency list pointers (elim-queue) and apply arithmetic operations similar to the original gauss elimination on the adjacency list.

With an simple example we can easily show how we adapt Gauss-elimination on adjacency list in a graph structure.

Lets say we have an equation like

$$0 \ 2 \ 0 \ 3 \ 5 \ 6 \ 7 \ 0 \ 0 \ 9 \ b = 4$$

and will eliminate an other equation in our linear sparse system like

$$0 \ 4 \ 4 \ 0 \ 0 \ 9 \ 0 \ 0 \ 1 \ 8 \ b = 6 .$$

These two equations have their first non-zero coefficient in the second column. So we can try to eliminate the second equation using the first equation. The ordinary result of gauss elimination will look like below.

$$0 \ 0 \ 4 \ -6 \ -10 \ -3 \ -14 \ 0 \ 1 \ -10 \ b = -2$$

But our goal is to deal only with non-zero coefficients so we can represent, as mentioned before, these two equations in adjacency list format. Now our problem will be look like as shown below :

$$(1,2)\rightarrow(3,3)\rightarrow(4,5)\rightarrow(5,6)\rightarrow(6,7)\rightarrow(9,9) \quad b \text{ value} = 4$$

(elim-list)

$$(1,4)\rightarrow(2,4)\rightarrow(5,9)\rightarrow(8,1)\rightarrow(9,8) \quad b \text{ value} = 6$$

(one adjacency list of the graph)

If we apply Gaussian elimination to these two link lists we will get

$$(2,4)\rightarrow(3,-6)\rightarrow(4,-10)\rightarrow(5,-3)\rightarrow(6,-14)\rightarrow(8,1)\rightarrow(9,-10)$$

with the b value = -2 as result linklist.

The elim-list looks for suitable adjacency lists in the whole graph and adds their pointers to the elim-queue. If the elim-list complete its search the adjacency list which is pointed by the head of the elim-queue is our new elim-list. The head of the elim-queue is removed and the adjacency list pointers which will be added by the new elim-list will be added to the end of the elim-queue. Each modified (eliminated) node is sendet to the neighbor server if no frame with the same first col value is sendet before. If not the adjacency list of this node is transformed to our frame structure used between the servers and sendet to the subsequent neighbor in the ring architecture. The elimination procedure terminates after the last elim-list finishes its search and do not add any new member to the elim-queue.

If the output of the elimination procedure is 2 and the first elim-list is a member of the elim-repeat set all members of the elim-repeat set is checked again.

3.2.3 Solve Procedure

Solving an nxn linear system requires two main steps. The first step is the transformation of the system using matrix operations into a upper traingular system and the second step is solve each x value with backsubstitution. The goal of transforming the system into triangular matrix is getting at least one row which can be solved.

This means that we gain a row which looks like this

$$a_{ij} x_j = b_i.$$

In this equation unknown x_j value can be easily solved (calculated).

$$x_j = b_i / a_{ij}$$

If we think about the same problem but use instead of the usual $n \times n$ matrix structure, a graph structure the solvability condition for any x_j will be the same but the representation will be different. If we represent

the $a_{ij} x_j = b_i$ equation in a graph structure then this equation can be symbolized as a node n_i which b_i value is b_i and has only one edge (neighbor) connected to the node n_j (Figure 3.5)

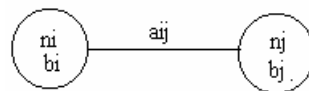


Figure 3.5 Graph representation of $a_{ij} x_j = b_i$

This can be also easily shown in adjacency list structure (Figure 3.6).

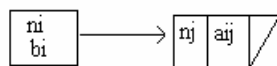


Figure 3.6 Adjacency list representation of $a_{ij} x_j = b_i$

After solving the x_j value, the edge which connect n_j to n_i and the node who has no neighbor anymore will be removed from the graph. This means that every solved x_i value shrinks the graph by one node. The solved x_i value is added to the x value linklist (x - list) which contains the set of solved x_i values.

Another step of our solve procedure is to visit every neighbor node n_i in our graph and find out if they have a neighbor node n_j with the id that is equal to an id of any x_j value in the x - list. If there is found such a neighbor node n_j the b_i value of node n_i is updated with

$$b_i = b_i - w_i * x_i$$

and the edge which connects n_i to n_j will be removed from the graph which will shrink the graph again. It is obvious that this step is essentially the operation of the back-substitution with one difference. The back-substitution is only done after making the system a triangular system.

Working with graphs supply us to minimize the requirement of a triangular system. Using the elimination procedure we force the system to get to a triangular system but the swap operation is not necessary. No back-substitution is done in our system. We try to shrink our graph whenever we can. X values which are solved in other workstations are received using the communication thread and are added to each workstations x list. So that we don't need to wait for a completion of the triangular system and every workstation begins to shrink its graph at the same time so we get a parallelism in the back-substitution part of our solving operation.

Each workstation sends its x-list if it solves any x values to its neighbor in the ring architecture.

Finding solvable x_i values in a graph takes less time as finding them in a $n \times n$ matrix. Only one step is required in a graph structure if this node has only one neighbor. Against to $n \times n$ matrix system. In $n \times n$ matrix systems the number of required if condition can be different from 1 to $n-1$ steps to find out if there is only one non-zero coefficient in the row.

4. Results

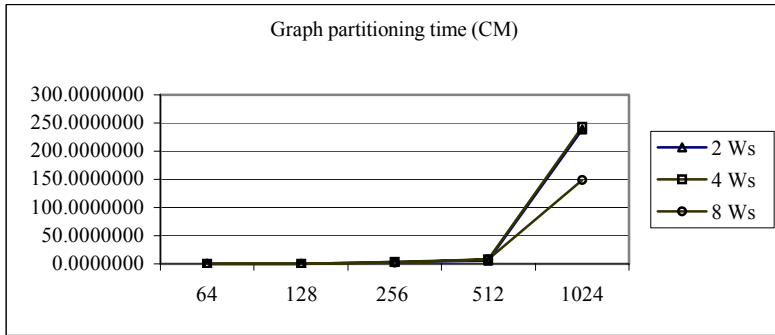
Our aim is to solve a sparse linear system faster using parallel methods. Our application has two main parts. The client side, one single processor which tries to parallelise our linear system $Ax = b$ data. And the server side, multiple processors (workstations) which are eliminate and solve x_i values. In this section we interpret the runtime results of both the server and the client processes. We compare runtimes of a serial algorithm and our parallel algorithm.

Parallelism is gained on the server side where the system is solved. The servers, which are communication in a ring architecture, try to solve the systems x values with a method based on the known Gauss- Elimination. We get the time results on 2,4 and 8 workstations. For every matrix size (64x64, 128x128, 256x256, 512x512, 1024x1024) we solve 50 linear systems for our isticitics. Our workstations are Sun Servers which server for the Ege University campus. One of the servers is an Ultra 10 (bornova.ege.edu.tr) with 256 MB RAM on it. The rest are Ultra 5 Sun Servers (didim.ube.ege.edu.tr,urla.ube.ege.edu.tr,bodrum.ube.ege.edu.tr,cesme.ube.ege.edu.tr,eng.ege.edu.tr,med.ege.edu.tr,agr.ege.edu.tr,madran.ube.ege.edu.tr) with 128 MB RAM. All Sun Servers run Solaris 7 as operation system.

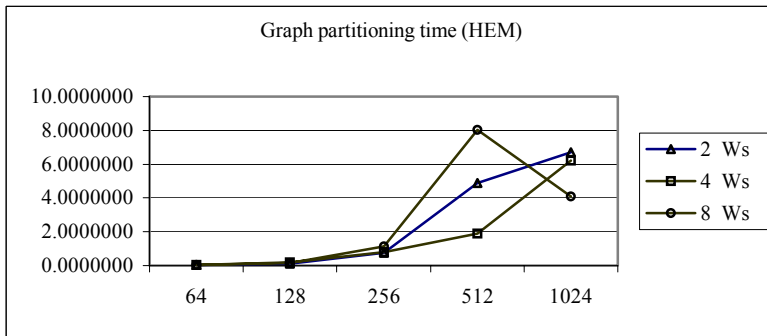
Lets begin to interpret the isticistics on the client side. (all time units are seconds).

4.1 Client Results

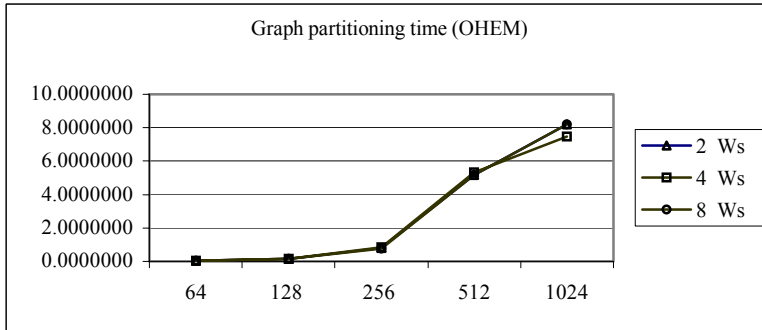
The client transfer the linear system to a graph structure. Partitioning of the graph is done using 3 different methods deccribed earlier (HEM,OHEM and CM). After partitioning the partitions are sendet to each corresponding workstation. Runtime values of each partitioning methods are shown in Figure 5.1



(a)



(b)



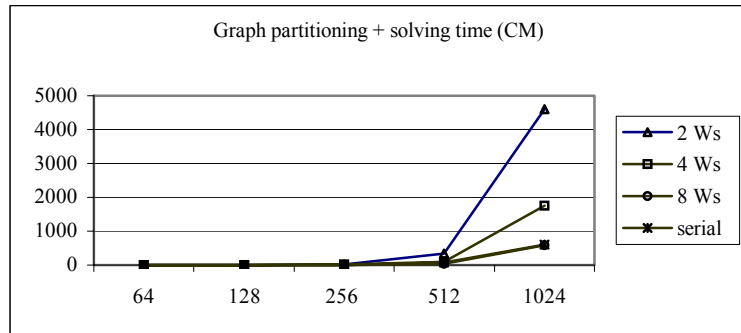
(c)

Figure 4.1 Graph partitioning times (a) CM (b) HEM (c) OHM

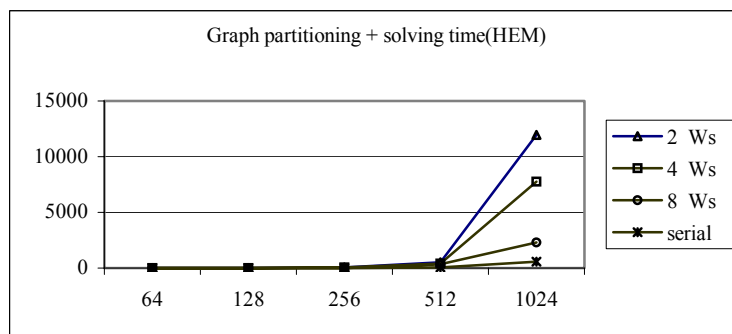
Figure 4.1(a) shows that CM spends the most time for graph partitioning. CM method apply on all nodes in the graph the BFS algorithm for finding center nodes. This operation takes an important amount of time. However the quality of the partitions created by CM method leads to less total system solving runtime. CM method creates more consistant and better partitions than HEM and OHM methods.

If we take a look at the total runtime values we can see that partitions created by CM have the best results. We define total runtime as the time spend for solving the system including the graph partitioning time.

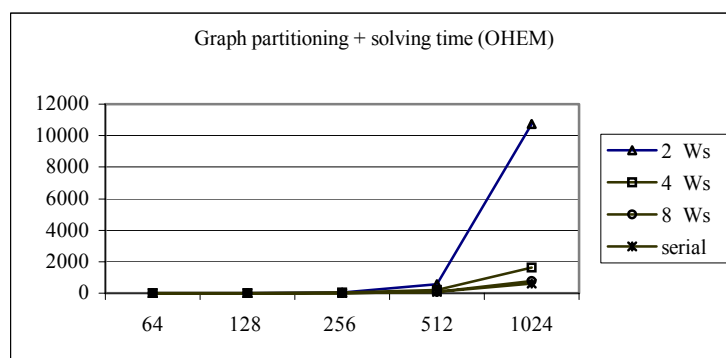
	64	128	256	512	1024
2 Ws	0.9548232	2.9325541	26.44362	334.4244588	4599.703757
4 Ws	1.1794134	2.13855132	12.1432349	101.4647568	1752.190474
8 Ws	1.96200784	2.16631088	7.2383755	40.97769992	<u>589.6194454</u>
seri	0.1050330	0.9378220	8.2239120	69.0343970	<u>601.1831570</u>



(a)



(b)



(c)

Figure 4.2 Graph partitioning + solving time (total runtime) (a) CM (b) HEM (c) OHEM

Average total runtime for matrix size 1024×1024 using 8 workstations is 589.619 sec (Figure 4.2(a)). The serial runtime is for the same matrix size 601.183 sec. We have an improvement of 10 seconds. If we only look at the part after the partitioning operation we will see that 440 seconds are spend to eliminate and solve the system in parallel. This is an improvement of 160 seconds.

CM method creates partitions which include approximately the same amount of nodes. The deviation related to the node number each server receives is near to 0. The deviation values gained using HEM and OHEM decrease with the increase of the workstation number but are too big to compare them with CM.

4.2 Server Results

Lets have a look at the server side of our protocol. For all three methods 2,4 and 8 Workstations are separately used to get the runtime results. The quality of the partitions effect directly the load balancing between the servers. The parameters we are going to examine in this section are the servers idle times, spend time for communication, the time spend for elimination and time spend for solving the x values.

We will separately study these values for 2,4 and 8 workstations.

4.2.1 Results using 2 Servers (Workstations)

Our goal is to balance the work. In the best case each of the two servers get partitions where the number of the nodes is approximately the half of the total node amount in the original graph. In this case forcing the system to a triangular system is mostly the job of the second server in the ring architecture.

The first server has for a matrix size of 1024×1024 and if partitions created by CM method are used a idle time ratio of %18, if HEM is used a ratio of %99 and for OHEM a ratio of %3 against the total solving time. The second server is nearly never idle. Especially in the HEM method we can see how much the quality of the partitions influence the idle time values.

The time spend for communication between the servers changes between 0.0001 sn and 0.4 sn depending on the used methods and the used matrix sizes.

For a matrix size of 1024×1024 the time spend for elimination ratio against the total solving time, is for the CM method %8, for HEM method %2 and for the OHEM method %3.

4.2.2 Results using 4 Servers (Workstations)

In the best case each server has to expect partitions where the amount of nodes in the partitions are approximately equal. We get the best partitions using the CM method so we expect before getting the results that the best runtime values will be for this method.

The idle times for each server and a matrix size of 1024x1024 are shown in Figure 4.3 for all methods.

	1024x1024 (HEM)	1024x1024 (OHEM)	1024X1024 (CM)
server 0	7726.00000	817.00000	822.44117
server 1	396.000000	227.00000	401.25000
server 2	<u>0.2136110</u>	<u>0.01563</u>	<u>4.14570</u>
server 3	<u>431.000000</u>	<u>0.00251</u>	<u>10.85676</u>

Figure 4.3 Sunucuların işsiz kalma süreleri

Figure 4.3 shows as that server₀ and server₁ have much more idle times than server₂ and server₃. Force the system to a triangular system is mostly done by the last two servers. This is because they spend less times idle. For the case where only 2 workstation are used the second server doesn't spend any time idle. For 4 workstations we see that the last two workstations have nearly no idle times.

The time spend for communication between the servers changes between 0.001 sn and 3.4 sn depending on the used methods and the used matrix sizes.

The 3. and 4. servers spend more time for elimination as for solving x values if the CM method is used. We can not say the same thing for the other methods. This value depend on the size of the partitions each server gets. The server which partition is bigger spend more time for elimination than the other servers. For a matrix size of 1024x1024 and partitions created by CM method

For a matrix size of 1024x1024 and partitions created by CM method the time spend for elimination ratio against the total solving time, is for server₀ %0.1, for server₁ %1, for server₂ %12 and for server₃ %8.

The time spend for solving shows opposite proportion to the time spend for elimination. For the same matrix size and the same method the the time spend for solving operation ratio against the total solving time, is for server₀ %43, for server₁ %24, for server₂ %17 and for server₃ %0.2.

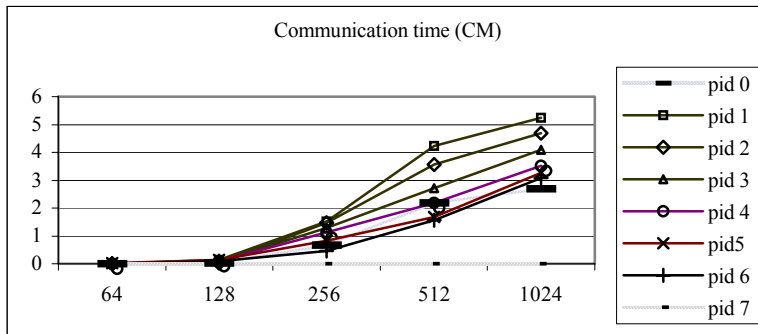
4.2.3 Results using 8 Servers (Workstations)

The results we get in the case where 8 servers are used are interesting for us. We begin to spend less than the serial runtime for solving the whole system.

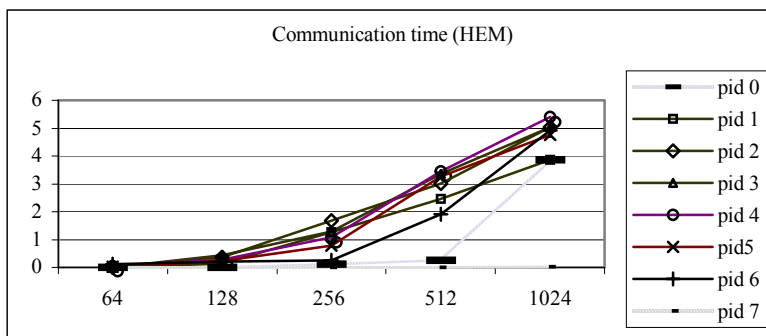
The minimum idle times for the servers is again gained using the CM method. The maximum idle time for all matrix sizes and CM method is 288 second, for HEM method 742 sn and for the OHEM method 637 second (for the first server).

If we could get our results using 16 servers we expect that these values will getting smaller.

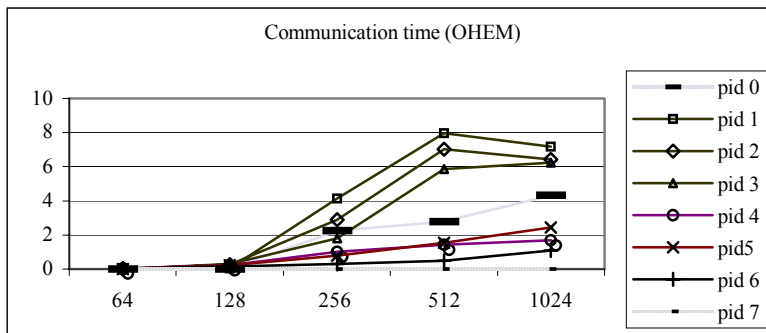
If we have a look at the communication times between the servers (Figure 4.4) we can again see that the best load balancing is gained by CM method (Figure 4.4(a)).



(a)



(b)



(c)

Figure 4.4 Communication times for 8 servers. (a) CM (b) HEM (c) OHEM

Like the results described before the time for elimination increase for server_i with the increase of the i value. The servers at the end of the ring spend more time for elimination and less time for solving the x values.

The best graph partitions are gained using CM method. The partition quality effect the solving time of linear systems and obtain smallest runtime values. However the CM method needs to assign some nodes in the graph as center nodes. Nodes which have at least a certain distance are assigned as center nodes. For that the BFS algorithm is applied to all nodes in the graph which makes the method expensive. OHEM and HEM methods supply their partitions relative faster than the CM method. A parallel method can be proposed to get a better runtime for CM method.

The partition quality of OHEM and HEM are not as balanced as the partitions

obtained by CM. But they are much faster because they do not apply the BSF algorithm on all nodes which is very expensive. Kerningham-Lin (KL) algorithm could be used if OHEM or HEM methods are preferred to refine and get more balanced partitions. Another way could be apply KL algorithm with gain calculation to the partitions of OHEM or HEM.

Lets look at the server side. Our protocol gives us the best runtimes if the number of nodes containing by the partitions are approximately equal.

The best partitions are gained using the CM method. We get the first faster runtime results according to the serial runtime using 8 workstations and a matrix size of 256x256. Runtime values for matrix sizes which are larger than 256 are again faster than the serial runtime. We didn't register any faster runtime results using OHEM and HEM.

The runtime results of OHEM and HEM are very incoherent. We apply our methods on 50 matrices for every workstation and matrix size combination and get similar runtime results if the partitions are created using the CM method. The load balancing is an important parameter for our runtime values.

The least idle time values and the best load balancing of our servers are again registered if CM method is used. During our tests we could see that occasionally partitions are created using OHEM and HEM in which nearly the whole linear system is given to only one server so that we can't talk about load balancing in any way. For these situations a protocol which not only relay on the client side for load balancing can be designed. So that even if he partitions are worse the protocol force the system to better load balancing by sending a part of one server to another which has less work.

We can also have look at the methods runtime performances ,nt the situation where only the workstation number changes. CM method make the best use of this increase. The partitions created by CM getting smaller with the increase of the workstation number so that each server has less to solve. The communication overhead doesn't very effect the runtime values.

The servers which spend much time for solving do spend less time for elimination. Our Matrix space consists of matrices which the number of non zero coefficients is about %20. Matrices which are more sparse could give us better results.

The Sun servers we use serve for the Ege University campus. We take our results in times where the amount of logged users is minimum but it is clear that better resluts could obtained if servers with no other job running on them are used instead of the servers.