ParLeda: A Library for Parallel Processing in Computational Geometry Applications

Mohammad Ghodsi *Mehdi Sharifzadeh*
Computer Engineering Department
Sharif University of Technology

ghodsi@sharif.ac.ir, shzadeh@yahoo.com

Abstract

ParLeda is a software library that provides the basic primitives needed for parallel implementation of computational geometry applications. It can also be used in implementing a parallel application that uses geometric data structures. The parallel model that we use is based on a new heterogeneous parallel model named HBSP which is based on BSP and is introduced here. *ParLeda* uses two main libraries that are widely used: MPI for its message passing in the parallel environment and LEDA for its data structures and computations. Dynamic load balancing and replicating C++ objects are two key features of this library. This library was implemented after a survey in researches on parallel computational geometry algorithms and selection of their common primitives.

Keywords: *Computational Geometry, Parallel Processing, Load Balancing, LEDA, MPI, ParLeda.*

1 Introduction

Researchers in many fields of science and engineering have a never-ending demand for more processing power and for increase in the computation efficiency. Computational Geometry (CG) problems with extensive amount of computation and huge input/output size are excellent candidates for parallel implementation. For example, DARPA Architecture Workshop Benchmark Study, inserted four computational geometry problem in the eleven problem list which they had provided for performance evaluation of parallel architectures [7].

Implementation of parallel CG applications is a quite time consuming job and needs good attention to many details. The purpose of this paper is to present issues involved in implementing a software library, called *ParLeda*, that provides a set of general parallel primitives to be used in parallel implementation of most applications with geometry data structures, specially CG applications. The primitives are selected such that the programmers are relieved from some details of the parallel implementation.

To select a good set of basic parallel primitives, we have studied efficient parallel algorithms used for different classical CG problems (such as convex hull, triangulation, etc.) and recognize their basic common parallel primitives. These primitives are then defined in a general setting and are implemented as programming API for the proposed software library.

The parallel computation model which *ParLeda* is based on is a heterogeneous model named Heterogeneous Bulk Synchronous Parallel (HBSP) which uses heterogeneous computation units in BSP[1] model.

Most parallel computational geometry algorithms use some computational phases which share common algorithmic behavior but differ in input data types. Most of these phases are not computational geometry specific and are used in another parallel algorithms too. In designing *ParLeda*, we have suggested an abstract definition for such phases which is data and algorithm independent. This idea has been taken from Morin's research in [4] where has suggested an API for a library named PLeda and has defined some basic parallel operators for computational geometry problems. However, the design and implementation of this library has been done from scratch.

As *ParLeda* works on a heterogeneous network of UNIX machines, we have designed and implemented algorithm specific load balancing methods in the library which will be explained later in this article. At the end of this paper, we will show a sample of programming with *ParLeda* API.

2 Previous Works

In this section we present some research on parallel CG problems we have considered in our survey.

Puppo, *et al.* developed a parallel algorithm for terrain Delaunay triangulation and implemented their algorithm on a CM-2 machine [12]. The problem had been addressed by several other authors in the literature but they have had an actual parallel implementation for the first time. Y. Ding and P.J. Densham [13] presented a parallel algorithm for constructing Delaunay triangulation which uses a dynamic, recursive and altering bisection approach to compose a rasterized space into partitions of which localized triangulation are constructed. The algorithm was implemented on a distributed memory transputer and the results were presented for a range of problem sizes.

G. Hristescu [10] addressed the problem of efficient parallel triangulation methods for a finite set of points in the plane and presented two approaches for the problem and implemented them on a hypercube. P. Magillo and E. Puppo [11] reviewed examples of parallel algorithms for different problems of terrain modeling and visualization. They have considered different programming paradigms and different architectures and have considered both the theoretical and practical aspects of this problem.

As an another research in parallel terrain modeling problems, Y. Ansel Teng, *et al.*[16] presented a parallel algorithm with $O(log^2 n)$ time complexity for computing the visible points of a polyhedral terrain from a given viewpoint. They improved the algorithm proposed by Katz, *et al.*

A. Clematis, *et al* [14] presented their experience in parallelising, in a systematic way, a class of Geographical Information Systems applications. They used PVM and Linda as communication libraries for spatial data handling. In a research article S. C. Roche and B. M. Gittings [15] discussed the effectiveness of both automatic and manual parallelising techniques in GIS applications. They have used these techniques in a polygon line shading algorithm and considered the results. M. J. Atallah and M. T. Goodrich [7] considered some well-known CG algorithms like

[1]Bulk Synchronous Parallel

convex hull, intersection of half-planes, kernel of a simple polygon, distance between two convex polygons, 3-dimensional maxima, and the visibility problem in the framework of parallelism. They have reminded that as many of CG problems arose in real time applications related to GIS, CAD/CAM, etc we need to solve them as fast as possible and for many of these problems, however, we already are at the limits of what can be achieved through sequential computation. Thus, it is natural to study what kinds of speed-ups can be achieved through parallel computing. M. J. Attallah in [5] studied some typical CG problems and the parallelisation of their best algorithms on parallel machine models like PRAM, Mesh, hypercube and some hybrid models. He stated that previous work in parallel CG had been mostly theoretical and only some researchers have developed special purpose parallel CG algorithms for special parallel machines.

As the first general experimental work on parallel CG, Patrik Morin [4] defined the API of a general parallel CG library to be used by CG programmers in order to develop parallel CG algorithms. He has called his LEDA [2] based library PLEDA.

In context of replicable objects that we use in our implementation, many ideas have been developed in order to provide parallelism for an object oriented language like C++. Their approach is based on implementing classes which can provide parallelism in their methods and can be used for sending and receiving data in a parallel object oriented environment. In this way, some libraries like Para++ [19] have been developed. Many of these libraries are in fact software shells on communication libraries like MPI or PVM and only some of them like Dome [20] provide a distributed environment for sending and receiving large data structures as vectors or arrays. Some approaches for parallelism has been developed in language structures. So the modifications have been done in the original language to generate a parallel programming language. As we know none of these approaches have primitives for communicating C++ objects between nodes of a parallel machine. Only some of them have features for special data types like arrays and lists [21, 22, 23].

3 *ParLeda* in Application Development

ParLeda has been designed after studying many classical parallel CG algorithms with the purpose of retrieving common basic primitives. The library provides basic primitives for partitioning the domain of problem and features for load balancing between computation units. *ParLeda*'s design and implementation concepts are platform independent and it is portable. MPI library, in situation of a message passing standard that has been implemented on several different platforms and portability is one of its design goals, plays a basic role in *ParLeda*'s functionality and prepares an environment for message passing over a TCP/IP based network. Some API functions of this library has been changed to be used in parallel CG applications.

ParLeda is also based on a a public domain software library called LEDA[2] [2]. which provides efficient implementation of many data structures and algorithms on CG and other common areas. Basic *ParLeda* primitives that provide data partitioning use LEDA's data types and can easily be used in data parallel programs that use LEDA data types as the building blocks of their data area. Developers in other relevant fields like GIS can use *ParLeda* in conjunction with their application specific libraries. In other words, one can build special purpose libraries for CG related applications like GIS or fluid dynamics applications over *ParLeda*. In the role of an interface, *ParLeda* provides parallel processing concepts (data partitioning, load balancing, etc.)

[2] Library of Efficient Data structures and Algorithms

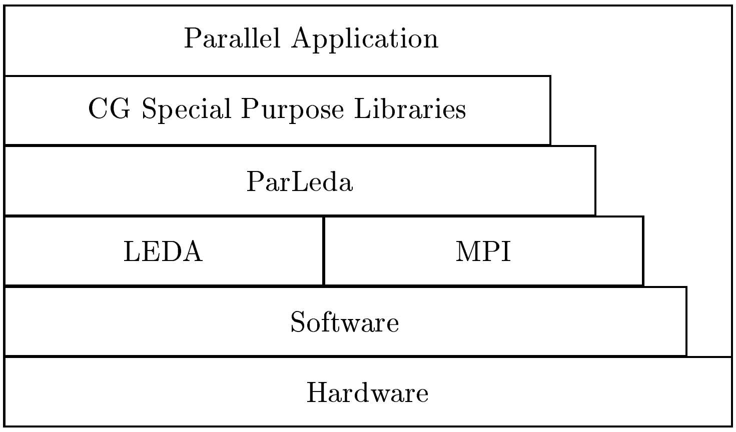

Figure 1: *ParLeda* in Parallel Application Development

and programmers can be relieved from the parallel implementation details and concentrate on application specific areas of the solution.

Figure 1 depicts a layer structure for developing a portable and modular parallel CG related application in *ParLeda*. Using separate related modules and an efficient implementation of the interfaces between layers can result in an efficient design of the solution.

4 Computational Model

ParLeda is based on a heterogeneous parallel computational model, named HBSP which is an extension to BSP model. The idea of design and using such model has been originated from [3] that used a similar model (HCGM) based on CGM. HBSP uses heterogeneous processors with different speeds. The speed of processors are considered as model parameters. This speed is a function of all software and hardware parameters like its virtual memory and processor speed involved in processor's overall performance. A Heterogeneous Bulk Synchronous Parallel (HBSP) machine has p different processing units P_0, P_1, ... P_{p-1} with different speeds of S_0, S_1, ... S_{p-1} which are integer numbers. The parameter $S = \sum_{i=0}^{p-1} S_i$ is denoted as the machine's total speed.

Each P_i processes a work with amount of W units, in time W/S_i. In this model, each processor is aware of the speeds of other processors. The speeds of the fastest and the slowest processor are denoted as S^{max} and S^{min} respectively.

In this way, $P^{max} = P_{\min\{i:S_i=S^{max}\}}$ and $P^{min} = P_{\min\{i:S_i=S^{min}\}}$ are identified as fastest and slowest processors. A typical HBSP with 4 processors has been showed in figure 2. In this machine, $P^{max} = P_1$, $S^{max} = 2$, $P^{min} = P_0$ and $S^{min} = 1$. In a special case that $S_0 = S_1 = \ldots = S_{p-1}$ the machines is BSP.

Two parameters g and l from BSP model are different in HBSP and should be defined as averages on different processing units in parallel machine. Each processor has its own local memory with a size dependent to its speed. As speeds of processors are not the same, a unified distribution of problem data causes an unbalanced processing load. So a good distribution

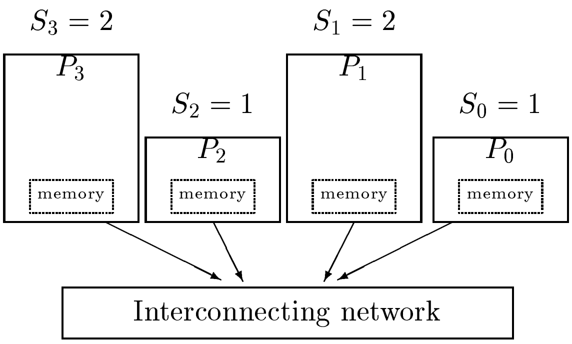

Figure 2: an *HBSP* with 4 processors

method should send more data to the faster processors to gain better performance. We will explain data partitioning methods for heterogeneous processors in section 5.

5 Data Partitioning/Moving Techniques

Most parallel CG algorithms are data parallel in nature. In these algorithms, partitioning of problem data set is an important issue and the relation between two neighboring data subsets and the size of each are as important.

In each partitioning method, a relation $\leq$ exists between each two neighboring partitions. This relation is defined based on domain type. As an example, when we divide 2-dimensional points into two right and left subsets with a vertical line, we use a relation $\leq$ for x coordinate of points.

For a balanced partitioning we should consider the size of each resulting subset. Some methods use regular shaped partitioning but others use unique data sizes in partitions which have no common shapes.

ParLeda provides data partitioning methods for common cases which we have observed in parallel CG problems. These methods are explained below. We have determined no shape for the area including selected geometric data sets in ParLeda, the only computed parameter is the size of partitions.

5.1 Random Sampling

Random parallel algorithms use random sampling as a efficient tool. Random sampling is choosing random subset with size $O(r)$ of a data set with size n. This subset is sent to one of the processors and that processor sends the results to other processors after the required computation. As P^{max} is the fastest processor in HBSP machine, it would be better to send the random sample to this processor. The algorithm is something like the following:

Algorithm 1

1. Each processor P_i chooses each data item in its local set for random set with the probability of $\frac{r}{n}$.

2. Chosen subsets in each processor are sent to one of them (P^{max}).

5.2 Linear Partitioning

If we assume that S is the set of problem data and the relation $\leq$ is a partial ordering on S, the linear partitioning of S is dividing it to p individual subsets $S_0, S_1, \ldots, S_{p-1}$ so that for all $x \in S_i$ and $y \in S_j$ that $i < j$ we have $x \leq y$.

Most parallel algorithms use linear partitioning as an initial stage. Sorting is a special case of linear partitioning which all subsets reside on one machine's local memory. The following algorithm for HBSP model is a typical linear partitioning method:

Algorithm 2

1. Each processor chooses a random sample of its data set with size $O(r)$ (r is a constant) using Random Sampling algorithm and sends it to processor P^{max}.
2. P^{max} sorts received subsets. We assume that the received data are inserted in sorted list $K_0, K_1, \ldots, K_{pr-1}$.
3. P^{max} computes $p+1$ splitters $s_0, s_1, \ldots, s_p$ among the list data items.

$$s_i = \begin{cases} -\infty & \text{if } i = 0 \\ K_{\lceil \sum_{j=0}^{i} r_j \rceil} & \text{if } 0 < i < p \\ \infty & \text{if } i = p \end{cases}$$

   r_j is the share of processor P_j of a data set with size pr.

4. P^{max} sends splitters to all other processors.
5. Each processor P_i puts data item x of its subset into bucket B_{ij} if and only if $s_j \leq x \leq s_{j+1}$.
6. Each processor P_i sends the data items in bucket B_{ij} to P_j.

5.3 Random Permutation

Random permutation is an approach for load balancing between processors of a parallel machine in which the problem data are sent to involved processors randomly. If the involved processors are similar, we should apply a homogeneous data distribution. But in HBSP uses the processor speeds in data distribution. The algorithm is:

Algorithm 3

1. Each processor P_i sends each data item to one of p buckets $B_{i0}, \ldots, B_{ip-1}$ in a random way. The probability of sending each data item to bucket B_{ij} is equal to $Prob_j$. ($\sum_{j=0}^{p-1} Prob_j = 1$)
2. Each processor P_i sends the data items in bucket B_{ij} to P_j.

The probability $Prob_j$ is computed based processor's work load. By means of a load balancing method from time complexity of the computation algorithm which will be applied on defined data sets, we can determine each processor work load. In this case the probability of sending data to a processor with a high work load is more than another processor.

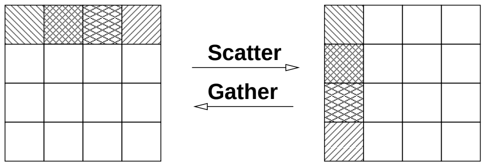

Figure 3: Scatter and Gather operators

5.4 Global Sorting

Sorting local data sets is the next step after partitioning of problem domain and distributing defined subsets. The following parallel sorting algorithm which is a combination of linear partitioning and sequential sorting and has been used in [10], is implemented in *ParLeda*.

Algorithm 4

1. Problem data is partitioned into p subsets using linear partitioning algorithm.

2. Each processor sorts its own data set in a sequential way and synchronous with other processors.

5.5 Scatter/Gather

Two dual primitives `scatter` and `gather` are used in distributing of a partitioning algorithm defined subsets and gathering local results by the root processor. In both operators one processor is in a special role named root and is responsible for sending and receiving of data sets. These operators are shown in figure 3.

6 Replicable Objects

LEDA has implemented the geometrical objects as C++ objects. So *ParLeda* must be able to exchange these objects between different processors in a parallel machine. As API functions of MPI are implemented in C, we can not use them to operate on C++ objects. In other words, this library and MPI library can not be used in packing, sending and receiving C++ objects and programmer should manually pack and send their data parts to other processors and rebuild them from their data parts in the target processor. In this way, we can restructure a new version of an object similar to original object only in data parts not in behavior.

As C++ objects have combined the methods and some control tables like inheritance table beside data parts, we can not use MPI transfer functions which have been implemented to work on basic data types. Some object properties like inheritance and polymorphism can not be moved from original object to its new version in another processor.

In contrast with suggested works, *ParLeda* has implemented a general approach and provided a procedure for converting a C++ class to an equivalent replicable C++ class. Programmer can drive his/her class based on a replicable class and define class virtual methods for packing and sending class objects to use MPI functions in order to send and rebuild objects in remote processors. He/She can set the number of sent buffers and their sizes to minimize the size of message including object data.

The main idea is that we should implement classes named replicable to provide data items needed for constructing a copy of an object based on an original object in their interfaces. These data is defined by class virtual methods. Two packing and rebuilding methods which are data independent and use these virtual methods have also been implemented. Any C++ class which is inherited from a replicable object and define its virtual methods can be used in MPI functions for sending and receiving in processors.

Every C++ object has a data part and some methods. For replicating this object on a different process or processor we need to know the data part and the dynamic part of its methods (we call theses parts extents). So if we have these data in the target process, we can rebuild the object. As an example for a LEDA `point` object, its x and y coordinates are the data we need to rebuild it. So for its replication or immigration (destructing original object) we pack its x and y coordinate, pack them and send them to the target process. In the target process we should unpack the received buffer, construct a point object and set its x and y coordinates to the data stored in the received buffer. This scenario is repeated for each object.

Now, we have a general replication method, but the programmer should need details about extents of the object which he want to replicate. If LEDA store a control data in its `point` class, replicated object that uses only x and y coordinate of the original point for its constructing is not the same as the original. So, we define two general classes for all replicable classes: One, basic replicable class which is replicable but have no replicable object in its data part. Two, composite replicable class which its data part is composed of replicable objects.

Each replicable class has virtual methods for the following: 1) providing information about extents and their sizes (ex. value of x(size=4bytes), value of y(size=4bytes)). 2) providing number of extents (ex. 2), and 3) pack and unpack methods which pack extents in buffers or unpack them from received buffers. The following example shows replicating a basic replicable object named `rpoint` in a process:

```
#include <PARLEDA/RPoint.h>
    point p(3,5);
    rpoint rp(p, MPI_COMM_WORLD);

    (void)rp.pack();
    MPI_Send(rp.pbuff(), rp.ppos(), MPI_PACKED, 1, BUFFER_SIZE,
             rp.getcomm());
```

Rebuilding object in the target process:

```
#include <PARLEDA/RPoint.h>
    point p;
    rpoint rp(p, MPI_COMM_WORLD);

    MPI_Recv(rp.pbuff(), rp.plen(), MPI_PACKED, 0, BUFFER_SIZE,
             rp.getcomm());
    rp.unpack();
```

It is interesting to know that in *ParLeda* each process replicates an object named Node-Info which has information about its running processor like its speed in other processes. This information is used for load balancing.

7 Load Balancing

In a parallel environment with homogeneous processors, partitioning of data domain into subsets of equal size will cause an equal load balance for processors. Examples of data domain can be the number of 2-dimensional points in a 2-d convex hull problem or the surface of partitions in a triangulation problem.

In a heterogeneous parallel machine, however, an equal size partitioning of data would result in unbalanced loads between processors. In this way, faster processors finish their work faster than the slower ones and should wait for possible synchronization.

Processor P with speed S can finish computation W in time $\frac{W}{S}$. (S is the parameter we considered in HBSP model definition) This time has been considered in *ParLeda* as a parameter for partitioning the problem domain. W is interpreted as the algorithm complexity of a computation step and is a function over n (problem input size) ($W = W(n)$). This parameter is defined for each processor based on its data subset and the computation needed for this subset. If all processors have the same value for this parameter, we hope that all of them finish computation W in the same time.

Given that P_i works on a data of size n_i and $W_i = W(n_i)$ is the time complexity of this work, in order to have all processors finish the work on their local data subsets concurrently and at the same time, we should have:

$$\frac{W(n_0)}{S_0} = \frac{W(n_1)}{S_1} = \ldots = \frac{W(n_{p-1})}{S_{p-1}}$$

For example in a parallel sort we should locally sort local data in each processor after distributing global data between processors. As the best time complexity of this phase is $W(n) = nlogn$ we should construct subsets of size n_1, n_2, ..., n_{p-1} so that:

$$\frac{n_0 log n_0}{S_0} = \frac{n_1 log n_1}{S_1} = \ldots = \frac{n_{p-1} log n_{p-1}}{S_{p-1}}$$

Since the amount of $\sum_{i=0}^{p-1} n_i = n$ and the values of S_i for each processor are known, we can solve this set of p equations and p variables and determine the value of n_i for each P_i.

Lemma 1 *In an HBSP machine with p processors $P_0, P_1, \ldots$, and $Pp-1$ with speeds S_0, S_1, $\ldots$, and $Sp-1$ respectively, a concurrent computation phase with time complexity $W = W(n)$ on a data of size n which is distributed between processors will be finished at the same time in all processors if and only if:*

$$\frac{W(n_0)}{S_0} = \frac{W(n_1)}{S_1} = \ldots = \frac{W(n_{p-1})}{S_{p-1}}$$

Proof:

We know that $\frac{W(n_i)}{S_i}$ is the time for completion of work W on data with size n_i in processor P_i which its speed is S_i. So the equation in the above lemma is clear.

In order to find n_i, we should solve this set of p equations and determine the value of p variables:

$$\begin{cases} \frac{W(n_0)}{S_0} = \frac{W(n_1)}{S_1} = \ldots = \frac{W(n_{p-1})}{S_{p-1}} \\ \sum_{i=0}^{p-1} n_i = n \end{cases}$$

n and S_i are fixed.

It is difficult to solve the set if $W(n)$ is a complex formula. So we use a way to determine values close to problem solution in *ParLeda*. Determining a range for the parameter $\frac{W(n_i)}{S_i}$ and using divide and conquer method are the main idea of the solution.

We assume that :

$$S_0 \leq S_1 \leq \ldots \leq S_{p-1}$$

So we have:

$$n_0 \leq n_1 \leq \ldots \leq n_{p-1}$$

It is clear that the following relation is true:

$$n_0 \leq \frac{n}{p} \leq n_{p-1}$$

As $W(n)$ is an incremental function of n so this relation is true too:

$$W(n_0) \leq W(\frac{n}{p}) \leq W(n_{p-1})$$

Now we can determine the range for value of $\frac{W(n_i)}{S_i}$:

$$\frac{W(n_0)}{S_0} \leq \frac{W(\frac{n}{p})}{S_0}$$

$$\frac{W(n_{p-1})}{S_{p-1}} \geq \frac{W(\frac{n}{p})}{S_{p-1}}$$

Or in another word:

$$\frac{W(\frac{n}{p})}{S_{p-1}} \leq \frac{W(n_0)}{S_0} = \ldots = \frac{W(n_{p-1})}{S_{p-1}} \leq \frac{W(\frac{n}{p})}{S_0}$$

The following divide and conquer algorithm finds the share of each processor in a balanced data partitioning using these two limits for parameter $\frac{W(n_i)}{S_i}$:

Algorithm 5

1. Initialize two parameters Pt_1 and Pt_p as the following:

$$Pt_1 = \frac{W(\frac{n}{p})}{S_{p-1}}$$

$$Pt_p = \frac{W(\frac{n}{p})}{S_0}$$

2. Initialize $Pt = \frac{Pt_1+Pt_p}{2}$.

3. Determine n_i for $i = 0, \ldots, p-1$ in relation $W(n_i) = S_i Pt$.

4. If $\sum_{i=0}^{p-1} n_i > n$ Then $Pt_p = Pt$ and follow the algorithm from step 2.

5. If $\sum_{i=0}^{p-1} n_i < n$ Then $Pt_1 = Pt$ and follow the algorithm from step 2.

6. In this step we have $\sum_{i=0}^{p-1} n_i = n$ and the value of n_i are the solution for our problem.

4 processors	2 processors	1 processor	number of points
10	15	21	2000
17	43	60	4000
26	62	90	6000

Table 1: The run time of parallel Delaunay triangulation using ParLeda (second)

8 Programming API

ParLeda has been based on a C++ class named `ParLeda` which has implemented all *ParLeda*'s functionalities as its methods. A programmer uses these functionalities with creating an object based on this class. Calling `ParLeda`'s methods is valid after calling its `Init` method and is invalid after calling `Finalize` method in source code. `Init` initializes *ParLeda* and `Finalize` gracefully shuts it down. Programmer should initialize *MPI* before calling `Init` with MPI API function `MPI_INIT`. *ParLeda* computes the speed of each processor using number of jobs in its run queue and some parameters like size of virtual memory and swap space for load balancing. Two methods `SetAlg` and `UnsetAlg` are used before and after data transmission phase. These methods set time complexity of the next computation step in *ParLeda*.

As an example, the following is a global sort which has been implemented using a serial sorting algorithm which time complexity of $O(nlogn)$:

```
ParLeda pl(MPI_COMM_WORLD);

void GlobalSort(list<int> data) {
  pl.SetAlg(PL_NLOGN);
  pl.Partition(data, int_cmp);
  pl.UnsetAlg();
  data.sort();
}
```

Algorithm's time complexity can be set in *ParLeda* using constants like `PL_NLOGN` which have been defined in the library.

9 Experimental Results

We have implemented a parallel Delaunay triangulation algorithm using *ParLeda* operators. The algorithm uses a method named "Dividing Wall" for partitioning the whole set of points in 2-dimensional space. In another phase, we determine the triangles of final triangulation which intersect with the dividing wall. Then we triangulate two partition resulted by the wall. The algorithm is a parallel master/slave algorithm in which a master process gathers final results from another processes.

In original version of the dividing wall algorithm separates the original set into two subsets of equal size. But using *ParLeda* load balancing operators we divide the set according to the appropriate processor loads.

It's interesting to say that *ParLeda* implementation of the algorithms is 20less that its original implementation in case of program lines.

4 processors	2 processors	number of points
2.1	1.4	2000
3.5	1.39	4000
3.46	1.45	6000

Table 2: The speed-up resulted by running parallel Delaunay triangulation using ParLeda

10 Summary and Conclusion

We have explained the process of design and implementation of a software library named *Parleda* for developing parallel CG applications. We have identified common primitives which are used in parallel CG algorithms. These operators can be used in computational data transmission and for the balanced distribution between processors. *ParLeda* has implemented typical processor data interchange methods. Programmers use a global object which has defined these methods in its interface and can dynamically balance processor loads. We have defined a parallel model for our parallel machine named *HBSP* which has heterogeneous processors.

References

[1] MPI Forum, "MPI: A message passing interface," *Proc. of Supercomputing '93*, 878–883, November 1993.

[2] K. Mehlhorn and S. Naher, "LEDA: A Platform for Combinatorial and Geometric Computing," 1994, `http://www.mpi-sb.mpg.de/LEDA.html`.

[3] P. R. Morin, "Two Topics in Applied Algorithmics," M.S. Thesis, School of CS., Carleton University, CA, 1998, Available through URL `http://www.scs.carleton.ca/ morin`.

[4] P. R. Morin, "PLEDA User's Manual (v0.0)," Personal Communication, 12 Dec 1997.

[5] A. Y. H. Zomaya (Ed.), "Parallel and Distributed Computing Handbook," McGraw-Hill, 1996.

[6] R. Healy (Ed.), "Parallel Processing Algorithms for GIS," Taylors & Francis, 1998.

[7] M. J. Atallah and M. T. Goodrich, "Deterministic Parallel Computational Geometry," 1993.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin, "Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers," *Proc. ACM 9th Annual Computational Geometry*, 298-307, 1993.

[9] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Khokhar, "A Randomized Parallel 3D Convex Hull Algorithm for Coarse Grained Multicomputers," 1993.

[10] G. Hristescu, "Parallel Triangulation of a Set of Point for Coarse Grained Multicomputers," Department of Computer Science, Rutgers University, October 1994.

[11] P. Magillo and E. Puppo, "Algorithms for Parallel Terrain Modelling and Visualization," Parallel Processing Algorithms for GIS, Taylors & Francis, 352-386, 1996.

[12] E. Puppo, LS Davis, D. DeMenthon, and A. Teng, "Parallel Terrain Triangulation," *International Journal of Geographical Information Systems*, **8**(2), 105-128, 1994.

[13] Y. Ding and P. J. Densham, "A Dynamic and Recursive Parallel Algorithm for Constructing Delaunay Triangulations," *Proceedings 6th International Symp. on Spatial Data Handling*, Edinburgh, UK, 682-696, 1994.

[14] A. Clematis, B. Falcidieno, and M. Spagnuolo "Parallel Processing on Heterogeneous Networks for GIS applications," *International Journal of Geographical Information Systems*, **10**(6), 747-767, 1996.

[15] S. C. Roche and B. M. Gittings, "Parallel Polygon Line Shading: The Quest for more computational power from an existing GIS algorithm," *International Journal of Geographical Information Systems*, **10**(6), 731-746, 1996.

[16] Y. A. Teng, D. Mount, E. Puppo, and L. S. Davis, "Parallelising an Algorithm for Visibility on Polyhedral Terrains," *International Journal of Computational Geometry and Applications*, World Scientific Publishing Company, 1995.

[17] L. De Floriani, C. Montani, and R. Scopigno, "Parallelizing Visibility Computations on Triangulated Terrains," *International Journal of Geographical Information Systems*, **8**(6), 515-531, 1994.

[18] Y. Ding and P. J. Densham, "Spatial Strategies for Parallel Spatial Modelling," *International Journal of Geographical Information Systems*, **10**(6), 669-698, 1996.

[19] "Para++: C++ bindings for Message Passing Libraries," EuroPvm'95, Sept. 1995, Lyon, FRANCE, `http://www.loria.fr/para++/parapp.html`.

[20] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan., "Dome: Parallel programming in a heterogeneous multi-user environment," Technical Report CMU-CS-95-137, School of Computer Science, Carnegie Mellon University, April 1995, `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/nectar-adamb/ web/Dome.html`.

[21] MPC++, `http://www.rwcp.or.jp/people/mpslab/mpc++/ mpc++.html`.

[22] Dennis Gannon, Shelby X. Yang, and Peter Beckman, "pC++", Department of Computer Science CICA Indiana University, Bloomington, Indiana, U.S.A., `http://www.extreme.indiana.edu/sage/`.

[23] Petitpierre C., "Synchronous C++, a Language for Interactive Applications", IEEE Computer, September 1998, pp. 65-72, `http://diwww.epfl.ch/w3lti/scpp.html`.