Simplex Method Implementation

CSE:501 Parallel Computing

Submitted by

Tulshi Chandra Das, Roll:BSSE0811

Maloy Kanti Sarkar, Roll:BSSE0834

Submitted to

Dr. B M Mainul Hossain
Associate Professor,IIT,DU

30 April, 2018

Abstract- The simplex method is frequently the most efficient method of solving linear programming (LP) problems. We review previous attempts to parallelize the simplex method in relation to efficient serial simplex techniques and the nature of practical LP problems. For the major challenge of solving general large sparse LP problems, there has been no parallelization of the simplex method that offers significantly improved performance over a good serial implementation. However, there has been some success in developing parallel solvers for LPs that are dense or have particular structural properties.
Keywords: linear programming, simplex method, parallel computing

1 Introduction

Simplex method is an algorithm which is used to solve the linear programming problems effeciently.
We consider the computation of econometric estimators given by the optimization problem: minx z = 8x1 +9x2, where x1,x2 are the variables.
2x1 + 3x2 ≤ 50
2x1 + 6x2 ≤ 80
3x1 + 3x2 ≤ 70
Simplex method works on standard form of the equations. So, we have to transform the equations to standard form: z = = 8x1 +9x2 + s1 + s2 + s3 2x1 + 3x2 + s1 = 50
2x1 + 6x2 + s2 = 80
3x1 + 3x2 + s3 = 70
Here s1,s2 and s3 are slack variable.
The Starting Tableau

Basic	z	x1	x2	s1	s2	s3	solution
z	1	-8	-9	0	0	0	0
s1	0	2	3	1	0	0	50
s2	0	2	6	0	1	0	80
s3	0	3	3	0	0	1	70

Figure1: Initial tableau of optimization

For optimization problems without closed form solutions, an iterative procedure is typically used. These algorithms successively evaluate the objective function z at different trial parameter vectors until a parameter vector achieves a convergence criterion. Let bi be the RHS of the ith row. Let aij be the coefficient of the entering variable xj in the ith row. The following “minimum ratio test” decides the leaving variable

The optimization steps are:

1. Determining the pivot row and pivot element using aij and xj
2. Make a non basic variable to basic variable
3. change the pivot row
4. Then change all other element in the table

The step 1 and 2 and 3 are serialized process and not costly. The step 4 is more costly and time consuming. We will parallelise the step 4.
There are two main sources of computation time in these numerical procedures. First, for each trial vector of parameters, there is the computational cost of evaluat-ing the objective function at this vector of parameters. Where the objective function involves simulation or the solution of a complex behavioral model, such as a dy-namic programming model, the objective function level computation costs can be large. A second source of computational costs is that for a high dimensional and continuous parameter space, a large number of vectors of parameters may need to be tried before the convergence criterion is achieved. This parameter level com-putation cost is increasing in the number of parameters, which is generally related to the complexity of the underlying behavioral model.

Related work

2.1 Concept of MPI

Most popular high-performance parallel architectures used in the parallel pro-gramming can be divided into two classes: message passing and shared storage. The cost of message passing parallel processing is larger, suitable for large-grain process-level parallel computing. Compared with other parallel programming en-vironment, message passing has good portability, supported by almost all paral-lel environments. Meanwhile, it has good scalability and complete asynchronous communication function, which can well decompose tasks according to the require-ments of users, organize data exchange between dierent processes and is applied to scalable parallel algorithms. MPI is an interface mode widely used in various parallel clusters and network environ- ments based on a variety of reliable mes-sage passing libraries. MPI is a message passing parallel programming standard used to build highly reliable, scalable and exible distributed applications, such as work ow, network management, communication services. MPI is a language-independent communications pro- tocol. FORTRAN, C and C++ can directly call the API library. The goals of MPI are high performance, scalability and portability. MPI is a library spec- ication for message passing, not a language. Message Passing Interface is a standard developed by the Message Passing Interface Forum (MPIF). MPI is a standard library specication designed to support parallel computing in a

distributed memory environment. The rst version (MPI-1) was published in 1994 and the second version (MPI-2) was published in 1997 [1]. Both point- to-point and collective communication are supported. MPICH is an available and portable implementation of MPI, a standard for message passing used in parallel computing. MPI has become the most popular message passing standard for parallel programming. There are several MPI implementations among which MPICH is the most popular one. MPICH1 is the original im- plementation of MPICH that implements the MPI- 1standard. MPICH2 is a high-performance and widely portable implementation of the MPI standard (both MPI-1 and MPI-2). The goals of MPICH2 are to provide an implementa- tion of MPI that eciently supports dierent computation and communication platforms including commodity clusters, high-speed networks and proprietary high-end computing systems. The standards of MPI are as follows [2]: -Point-to-point communication
-Collective operations
-Communication contexts
-Process groups.
-Process topologies
-Bindings for FORTRAN 77 and C
-Environmental management and inquiry
-Proling interface

2.2 MPI Functions

MPI is a library with hundreds of function-calling interfaces, and FORTRAN, C language and C++ can directly call these functions. Many parallel programs.

Table 1: MPI Basic Functions

Function	Functionalities
MPI Init()	Initialization
MPI Finalize()	Termination
MPI Send()	Send
MPI Recv()	Receive
MPI Comm size()	Access to the number of processes
MPI Comm rank()	Access to the identication number of a process

can be written with just six basic functions, almost complete all of the communication functions. Table 1 illustrates the basic functions. MPI Init() initializes the MPI environment and assigns all spawned processes; MPI Finalize() termi-nates the MPI environment; MPI Comm size() nds the number of processes in a

communication group; MPI Comm rank() gives the identication num- ber of a process in a communication group; MPI Send() sends message to the destination

process of rank dest and MPI Recv() receives message from the specied process of rank source.

2.3 The Messing Passing Process of MPI

MPI is a parallel programming standard based on message passing, whose func-tion is to exchange information, coordinate and control the implementa- tion steps with the denition of program grammar and semantics in the core library by send-ing messages between the concurrent execution parts. First all of the MPI programs contain mpi:h header le, and then complete the initial- ization of the program by MPI Init(), after that, establish process topology structure and new communicator and call the functions and applications to be used for each process, nally use MPI Finalize() to terminate each process.

2.4 Simplex algorithm

Suppose, Objective function,maximise z = ax1 + bx2 Subject to constraints,

$ax1 + bx2 \leq c1$

$ax1 + bx2 \leq c2$

$ax1 + bx2 \leq c3$

$x1, x2 \geq 0$

Introducing slack variables, the LPP is same as, Objective function,maximise z =

$ax1 + bx2 + s1 + s2 + s3$

$ax1 + bx2 + s1 = c1$

$ax1 + bx2 + s2 = c2$

$ax1 + bx2 + s3 = c3$

$x1, x2, s1, s2, s3 \geq 0$

Basic Feasible Solution

Let,

-x1,x2 as non basic variable

-Then BFS can be got immediately as ,(0,0,c1,c2,c3)

This is possible because in the LHS of the constraint equations (*) the coeffi-cients of the basic variables s1, s2, s3 form an Identity matrix (of size 3 x 3).

Closer Look (1)

We note that the z-row is the objective function row.

The remaining 3 rows are the basic variable rows.

Each row corresponds to a basic variable;

– the leftmost variable denotes the basic variable corresponding to that row.

In the objective function row,

– the coefficients of the basic variables are zero.

In each column corresponding to a basic variable,

– basic variable has a non-zero coefficient, namely 1,

– all the other basic variables have zero coefficients.

We see at present, the objective function, z, has value zero.

– the leftmost variable denotes the basic variable corresponding to that row.

We now seek to make

– one of the nonbasic variables as basic (and so) one of the basic variables will become nonbasic (that is will drop down to zero)

– the nonbasic variable that will become basic is chosen such that the objective function will "improve":

The nonbasic variable that will become basic is referred to as "entering" vari-able and the basic variable that will become nonbasic is referred to as "leav-ing" variable.

Criterion for "leaving" variable (Feasibilty Condition)

Let while b_i be the RHS of the ith row. Let a_i j be the coefficient of the entering variable x_j in the ith row. The following "minimum ratio test" decides the leaving variable: Choose x_k as the leaving variable where k is given as that row index i for which the ratio

$\{b_i \div a_i\ j : a_i\ j \geq 0\}$ is least (break the ties arbitrary)

Pivot element:

- The entering variable column is called the pivot column.
- The leaving variable row is called the pivot row.

- The coefficient in the intersection of the two is referred to as the pivot ele-ment.

Operations: We apply elementary row operations to modify the simplex tableau so that the pivot column has 1 at the pivot element and zero in all other places.

The elementary Row operations are as follows:

- New pivot row = old pivot row pivot element
- New z row = old z row – (coefficient of the entering variable in old z row * New pivot row)
- Any other new row = corresponding old row – (old coefficient of the entering variable in that row * New pivot row)
- We shall also change the legend of the new pivot row only as the entering variable.

The last tableau is the optimal tableau as all entries in the objective function row

are ≥ 0 and the LPP is a maximization problem.

3 Methodology:

In this section, we will propose an implementation of a parallel Simplex method based on MPI. Before introducing our Simplex algorithm, we first give a short description of the Sequential Simplex algorithm in Section *3.1*.

3.1 Sequential Simplex Method Algorithm:

In general, simplex method is implemented with serial code. Each step of update table or data is sequential and straightforward. The steps are given below:

Step 0: (Initialization):
Start with a feasible basic solution and construct the corresponding simplex tableau.

Step 1: (Choice of entering variable):
If $a_{0j} \geq 0$, j=1, 2,….,*n*, STOP. The current solution is optimal. Otherwise, choose the entering variable using the following pivoting rule: $a_{0s} = \min \{a_{0j} : a_{0j} < 0, j = 1,2, \ldots, n\}$

Step 2: (Choice of leaving variable):
Let I = {i : $a_{is} > 0$}. If I=∅, STOP. The problem (LP.1) is unbounded. Otherwise, choose the leaving variable using the minimum ratio test:

$$\frac{\text{br}}{ars} = \min\left\{\frac{\text{bi}}{ais} : i \in \text{I}\right\}$$

Step 3: (Pivoting):
The pivoting element is the variable a_{rs}. Construct the next simplex tableau as follows:
Let, $a_{rj} \leftarrow \frac{\text{arj}}{ars}$, $\text{j} = 1,2, \ldots, \text{n}, \text{n}+1$
And

, $a_{ij} \leftarrow aij - \frac{arj}{ars} ais,\ i = 0,1,2,\dots,m, (i \neq r)$, j=1,2,….n,n+1
Go to step1.

The simplex algorithm uses the Gauss-Jordan transformation of the tableau to move from one basic feasible solution to another. Each iteration of the simplex algorithm is relatively expensive. This can be seen by examining the previous formal description of the simplex algorithm. More precisely, the number of multiplications and additions at each iteration is approximately equal to $m(m\text{-}n)+n+1$ and $m(nm+1)$ respectively, where m is the number of constraints and n is the number of variables. This happens whenever n and m are large, in which case nearly 100% of the cpu-time is spent in Step 3 (Pivoting). In this third Step, a multiple of row r is added to row I, (this is the only double nested loop executed at each iteration).[3]

3.2 Parallel Simplex Method Algorithm with MPI:

We implemented the simplex method using a straightforward application of the C language tools and the Message Passing Interface (MPI). The storage of the simplex tableau was carried out using a (0:m)×(0:m+n) dimensioned array. Also, in order to allocate the work across multiple processors, MPI was used. The coefficients of the objective function are represented using the q vector, the number of processors is denoted by NPRS, the current tableau is denoted by TABL and finally the rank of processor is denoted by ITSK.[4]

Parallel Simplex Algorithm Begin

1- for 0 ≤ i < m+ n do
In processor 0: Set *q[i]:=TABL[0][i]*
for 0 ≤ k < *NPRS* pardo
{for 0 ≤ i < *m/ NPRS* do
for 0 ≤ j < m+ n do
Set *C[i][j]:=TABL[k*m/ NPRS+i]*
for 0 ≤ j < *m/ NPRS* do
Set *b[j] := TABL[j][M+N]*
}

2- In processor 0
for 0 ≤ i < n do
search i (where *q[i]<0*) Set *column := i*
if failure Goto (10).

3- for 0 ≤ k < *NPRS* pardo
X:=min(b[j]/C[j][column], 0 ≤ j < *m/ NPRS)*
Set row_no:=j
4- for 0 ≤ k < *NPRS* pardo
Send (*X,ITSK*) to processor 0
In processor 0
Search *ITSK* which corresponds to *min (X)*
min_L:=ITSK
5- From processor *min_L*
Send row *g:=C[row_no][:]* to all processor
Send variable *h:=b[[row_no]* to all processor

6- for 0 ≤ k < *NPRS* pardo
for 0 ≤ i < *m/ NPRS* do{
Set *a:=C[i][column]/g[column]*
for 0 ≤ j < m+ n do

Set $C[i][j]:=C[i][j]-a*g[j]$
Set $b[i]:=b[i]-a*h$
}

7- In processor min_L
for $0 \le j < m+n$ do{
set $C[row\_no][j] := g[j]/g[column]$
set $b[row\_no] := h / g[column]$
}

8- In processor 0
$a:=q[column]/g[column]$
For $0 \le i < m+n$ do {
Set q[i] := q[i]-a*g[i]
Set q[n+m] := q[m+n] - a*h
}

9- Goto (2)

10- **F**or $0 \le i < m+n$ do
In processor 0
Set $TABL[0][i] := q[i]$
for $0 \le k < NPRS$ pardo{
for $0 \le i < m/NPRS$ do
For $0 \le j < m+n$ do
Set $TABL[k*m/NPRS+i][j] := C[i][j]$
for $0 \le j < n/NPRS$ do
Set $TABL[j][m+n] := b[j]$
}
End.

4 Experiments

In this section, we first give the experimental environment in Section *4.1*. Second, we give the description of experimental data sets in Section *4.2*. Last, we give the experimental results and analysis in Section

4.1 Experimental Environment:

The hardware platform in this paper uses four PC. The software environment uses the following configuration.

1. Ubuntu 14.04 LTS:
Memory: 3.8 GiB
Processor: Intel ® Core™ i5-3470 CPU @ 3.20GHz × 4
OS type: 64-bit
Disk: 107.1 GB
Graphics: Intel ® Ivybridge Desktop

2. Ubuntu 16.04 LTS:
Memory: 3.7 GiB
Processor: Intel ® Core™ i3 CPU 540@ 3.07 GHz × 4
OS type: 64-bit
Disk: 44.2 GB
Graphics: Intel ® Ironlake Desktop

3. Ubuntu 16.04 LTS:

Memory: 3.8 GiB
Processor: Intel ® Core™ i5-3470 CPU @ 2.20GHz × 4
OS type: 64-bit
Disk: 51.5 GB
Graphics: Intel ® Ivybridge Desktop

4. Ubuntu_14.04 LTS:
Memory: 3.8 GiB
Processor: Intel ® Core™ i3-3470 CPU @ 2.20GHz × 4
OS type: 64-bit
Disk: 51.5 GB
Graphics: Intel ® Ivybridge Desktop

In terms of aforementioned platform, sublime text 3 is used to develop procedures. Considering the fairness of comparison, the configuration of MPI parallel development platform is based on open resource project and the experimental platform has a gcc complier based on MinGW (Minimalist GNU for Ubuntu).

4.2 Data Sets
Experimental data sets are selected from the github.com Dataset Repository.

4.3 Experimental Result and analysis:
In our experiments, the time cost is the key performance.

Table 2: Description of Dataset

Code number of Data Set	Name of Data Set	Size(KB)	Number of instance
1	afiro	4	28
2	adlittle	17	57
3	agg	98	489

Table 3: Comparison results within processes

Dataset No.	Execution Time(microsecond)				
	Serial	Clustered with 2 node(computer)			
		process			
		4	8	12	16
1	1593	530696	569165	10018335	9687363
2	83822	10432230	11364384	Waiting state	Waiting state
3	1060958	8809797	9699020	Waiting state	Waiting state

Table 4: Comparison results within processes

Dataset No.	Execution Time(microsecond)			
	Clustered with 3 node(computer)			
	process			
	4	6	8	12
1	807983	870251	783543	1824435
2	16918492	17174850	15764538	38181190
3	12692963	13821337	12305146	25850273

Table 5: Comparison results within processes

Dataset No.	Execution Time(microsecond)				
	Clustered with 4 node(computer)				
	process				
	4	8	12	16	20
1	1150373	1076964	1675929	1144053	10053268
2	19987390	19157692	32246914	20105473	Waiting
3	15608005	14547605	22298906	15017849	waiting

In the table3 we show results of processes with cluster of 2 pc and the serialized process. In table4 we show the experiment result in cluster of 3 pc and the table5 show the result of experiment clustering of 4 pc. We have tested the result with several processes. We have calculated the speed up factor of each process using the rule:

Speed up $=\frac{\text{execution time of serial process}}{execution\ time\ of\ parallel\ process}$

Table 6: Comparison of speed of processes corresponding to serial process

Dataset No.	Speed up			
	Clustered with 2 node(computer)			
	process			
	4	8	12	16
1	0.003	0.0027	0.00018	0.00016
2	0.008	0.0007	Waiting state	Waiting state
3	0.12	0.1	Waiting state	Waiting state

Table 7: Comparison of speed of processes corresponding to serial process

Dataset No.	Speed up in 3 node cluster			
	Clustered with 3 node(computer)			
	process			
	4	6	8	12
1	0.02	0.018	0.02	0.008
2	0.004	0.004	0.005	0.002
3	0.08	0.07	0.086	0.04

Table 8: Comparison of speed of processes corresponding to serial process

Dataset No.	Speed up				
	Clustered with 4 node(computer)				
	process				
	4	8	12	16	20
1	0.001	0.001	0.009	0.001	0.0001
2	0.004	0.004	0.002	0.004	Waiting
3	0.06	0.07	0.047	0.07	waiting

We find that in each process, speed factor is influenced by data set, number of process, communication overhead and number of clustering node. We find the increase of speed up according to size of data set. Speed up also increases according to clustering nodes. For small dataset number of clustering node don't contribute to speed up enough. We see for larger dataset clustering affect a positive sense to speed up. If the number of process increases upto a bound, processes tend to enter in a waiting state. If our dataset is larger and the number of clustering is enough then we can achieve a better speed up.

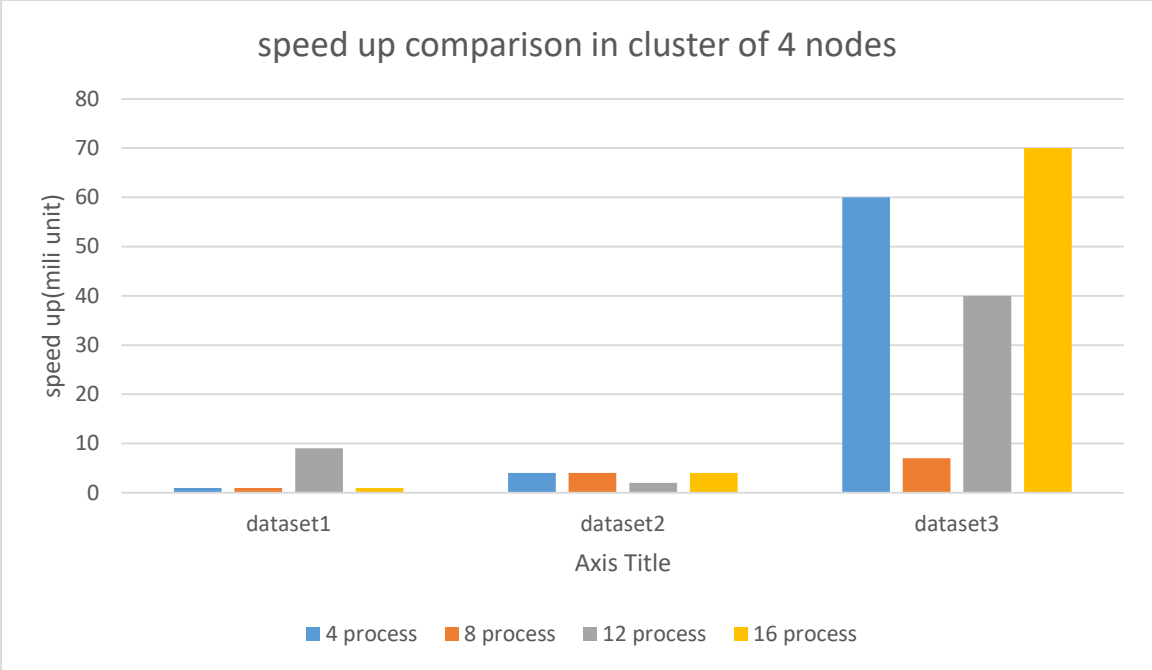

Figure2: Comparison of speed up in different dataset with processes

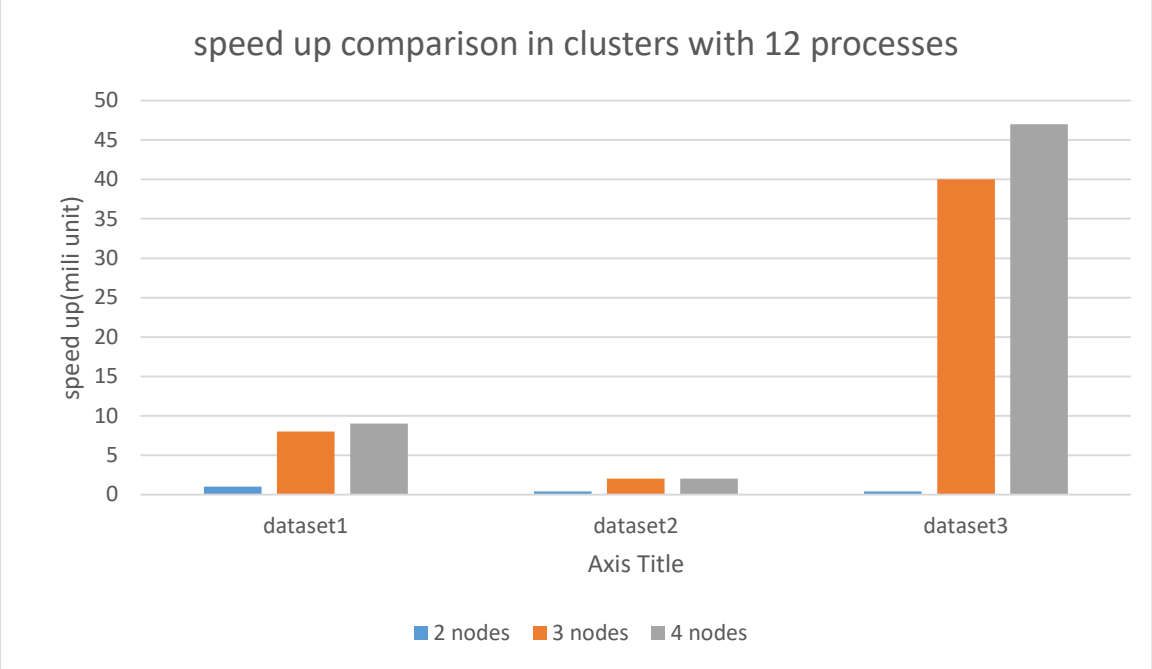

Figure3: Comparison of speed up between clustering nodes with 12 processes

5. Discussion

5.1 Performance Defects

From the performance analysis figures given in experiment and analysis part, we see that speed up is influenced by process number and clustering node and dataset size. The more dataset is large enough the more speed is acquired by the clustering. The execution time of serial code is for dataset1 is 1593 microsecond. For 2 nodes the execution time of MPI program is 530696 for 4 process. So, we see that there is no performance gain. Speed up increases for larger data set respective to clustering node. But if number of processes is not enough respective to clustering node performance don't increase enough. In figure 2, see the most speed up for 16 processes in cluster of 4 nodes in dataset 3. In dataset1 and dataset2 clustering doesn't affect enough. In some cases, program enter into waiting state. It happens when there are more processes than the clustering nodes. In this case, some process can't get process to run, so it get in waiting state.

6 Conclusion

We proposed a parallel Simplex method based on MPI (called MKmeans) in this paper. Meanwhile, the configuration of MPI parallel development platform based on open resource MPICH in Ubuntu is implemented, whose ideas and methods can be ported to Windows or other platforms. Experimental results show that the parallel implementation of the Simplex method is efficient in the clustering on large data sets.

Acknowledgment

At first, we thank almighty God to help us to complete the project. We wish to thank to our course teacher Dr. B M Mainul Hossain, Associate Professor, Institute of Information Technology, University of Dhaka for his valuable direction and help to complete our paper.

References

1. W. Gropp, E. Lusk, "Implementing MPI: the 1994 MPI Implementors Work-shop", Pro- ceedings of Scalable Parallel Libraries Conference, pp. 55-59, 2002.

2. Y. Aoyama, J. Nakano, "RS/6000 SP: Practical MPI Programming", Interna-tional Tech- nical Support Organization,August, 1999.

3. World Academy of Science, Engineering and Technology 23 2006

4. World Academy of Science, Engineering and Technology 23 2006