Coursework I

Parallel Computation (CSC 305)

Name: Stefan Dirnstorfer
Student no. 968048870

Date: 6. January 1996

1 The algorithm

The algorithm for the calculation of eigen vectors consists mainly of repeated matrix vector multiplication. The procedure can be summarized by the following formula:

$$\begin{aligned} v_0 &= \text{initial vector} \\ l_n &= \|v_n\| \\ v_{n+1} &= \tfrac{1}{l_n}\mathbf{A}\, v_n \end{aligned}$$

This is a slight variation of the original algorithm. But the changes are only achieved by reordering the steps, which saves one extra loop for resizing the vector and one point of communication between the processes. The only difference in the result is that the final vector does not have the norm one but the norm of its eigenvalue, which does not affect the fact weather it is an eigenvector or not. The actual eigen vector is the limit of v_n:

$$v = \lim_{n\to\infty} v_n$$

To determine weather the limit is reached close enough the changes in the norm of the vector is considered. If the difference of the norms falls below a certain tolerance the calculation terminated.

2 Using threads on newton

The newton computer is a shared memory parallel machine. Its operating system allocates the processor to certain threads. To ease the creation and the usage of threads a package "Encore Parallel Threads" is used to compile the source.

2.1 The program code

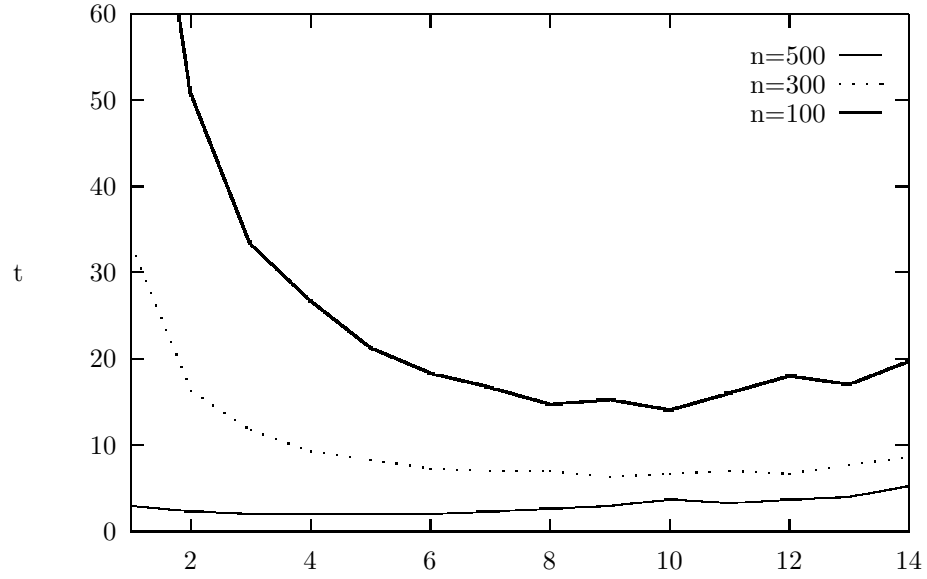
The program was derived from the given example program for the parallel calculation of the vector dot product. The main changes are the new data structure and the code for the child process which now involves synchronization. The three variables "A", "x" and "y" contain the matrix, the start vector v_0 and an array of two vector which is used to store temporary data during the calculation. The integer "curry" indicates which of the two vectors in y is currently used to store the new results in. This makes it possible to use an other vector for every result without copying it to the original variable at every step. The variables "norm" and "lastnorm" are used to calculate the norm of the current vector and to save the norm of the previous calculation. The "lastnorm" is initialized with one, the norm of the first vector v_0. The variable "proccount" is used to count the number of processes that finished their part of computation. It is initialized to the total number of processes and decreased by every child until the last child resets it to the initial value and work can start again. The parent function initializes all these values, copies the start vector in "x" into the "y" buffer, distributes the work, starts the children and copies their result back to "x". The child function processes the data allocated to it, by repeated scalar multiplication. The biggest result of these multiplications is stored in the variable "maximum". After that the global norm is updated. Therefore the monitor "controlFlag" is entered, the norm updated and the process counter reduced. If counter reaches zero i.e. the last process updates the norm several variables have to be reseted before the next step can be started and the currently used vector in "y" (i.e. "curry") has to be changed. After that the monitor is exited after signalizing the next process to proceed working. If the counter "proccount" was equal to one, i.e. no other process is waiting any more, no signal is sent. After all processes are released, the next step of calculation is started, provided the termination condition is not reached.

2.2 Testing

Here is an example run with a four dimensional matrix

```
 n = 4, p = 5
          -8           2          -8           2
           2          -3          -8           7
          -8          -8          -8          -8
           2           7          -8          -3
The Eigenvec is:
     12.1491     5.45971     18.1612     5.45971
The Eigenvalue is:
18.1612
Elapsed time:
1 sec
```

The following graph show how the calculation time of a fixed problem size changes with the number of processes. The times are averages over three runs.



For the speedups and efficiencies we get the following values

n	single processor	fastest parallel	number of proc.	speedup	Efficiency
100	3 sec	2 sec	3	1.5	50 %
300	33 sec	6.3 sec	9	5.2	58 %
500	96 sec	14 sec	10	6.8	68 %

2.3 Symmetric matrices

If a symmetric matrix is processed a great deal of memory can be saved. The total amount of required space for the matrix is $n \times (n+1)/2$ times the space of one number. If we decide to store the data by the rows of the lower triangle in an array "A" we get the following formula for accessing a certain element.:

$$a_{i\,j} = \begin{cases} A[i \times (i+1)/2 + j] & \text{for } j \leq i \\ A[j \times (j+i)/2 + i] & \text{for } j > i \end{cases}$$

In order to increase efficiency the loop that runs through the elements of the matrix can be divided into two pieces for which either of the two formulas applies.

3 Using PVM

This program runs the eigen vector algorithm on a distributed memory parallel machine, which in this case is an network of several Linux computers.

3.1 The source code

The source code consists of two parts. The master that initializes and distributes the data and the slaves that just compute their part of the calculation and sends back the results.
The source code for both is derived from the example for parallel vector scaling. After initializing the matrix and the initial vector "x" the first block is sent to every slave process It contains the dimension, the number of the first and the last row that is to be processed and the necessary part of the matrix is sent. This information is only sent once, since it does not change during run time. The main loop runs until the difference between the norm of the current vector and the norm of the previous vector reaches the demanded precision. In this loop the non constant data is sent, which includes the current vector "x" and its norm. After that the results are expected from every slave, i.e. the processed part of the vector and the norm of this part of the vector. The greatest value of these norms is the norm of the whole vector. When the loop reaches its breaking condition the calculated values are printed and all slave processes are killed.
The slave just receives the data as it is sent by the master and does the actual calculation in an endless loop until its process is killed by the master.

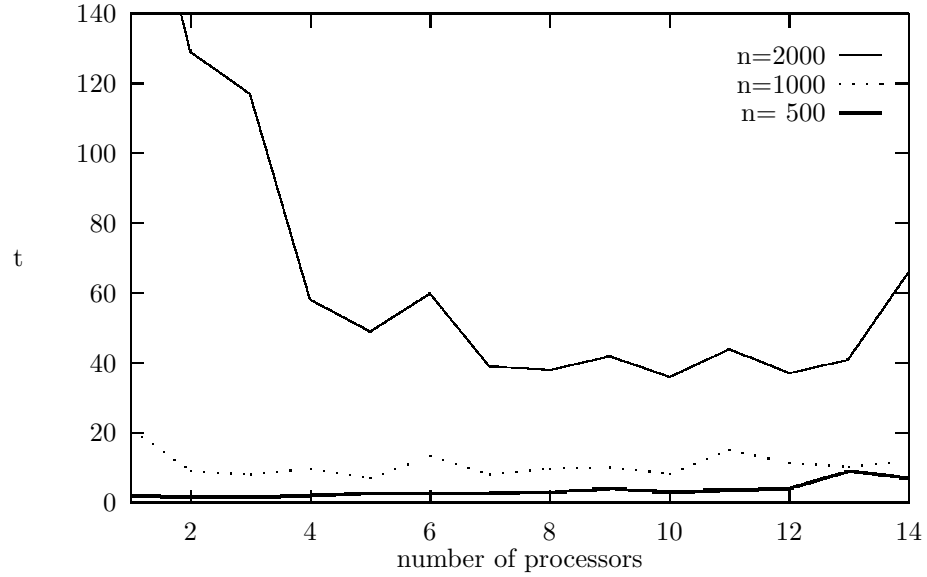
3.2 Testing

Here is an example run:

```
pvm> spawn -> master
[4]
1 successful
t140009
[4:t140009] n = 4, p = 5
[4:t140009]            0          -5         -10           5
[4:t140009]          -10           8           6           4
[4:t140009]            0           1           2           3
[4:t140009]          -10          -6          -2           2
[4:t140009] Eigenvector is:
[4:t140009]     -6.18964     12.7475    0.767645    -1.49878
[4:t140009] Eigenvalue is:
[4:t140009] 12.7475
[4:t140009] Elapsed time:
[4:t140009] 6 sec
[4:t180009] EOF
[4:t1c0009] EOF
[4:t200008] EOF
[4:t240007] EOF
[4:t280006] EOF
[4:t140009] EOF
[4] finished
```

The following graph shows the calcuation time. The times are averages over three runs. As one can see at the small problem sizes the calculation time depends only

slightly on the number of processes since the time is determined by the communication instead of calculation. But at a matrix dimension of 2000 the situation changes completely. At a number of four processes the calculation time drops rapidly. The only reasonable explanation for this is that at this point the matrix of a total size of 16MB is broken down into pieces small enough to fit into the memory without swapping. Therefore the main advantage of this distributed memory system is the greater memory not the calculative power.



3.3 Symmetric matrices

The storage of the matrix in the master code can be reduced as described in 2.3. The rows of the matrix can be expanded when they are sent to the slaves whose code stays unchanged.
If the storage of the slave should also be minimized a very fancy algorithm is required, which would lead to much more communication.

4 Source code

4.1 newton

```
#include <Thread.h>
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
const int n=4;
const int priority = 2;
const int kiloByte = 1024;
const int megaByte = kiloByte*kiloByte;
const int stackSize = 20*kiloByte;
const int dataSize = 2*megaByte;
int nprocs;
const double precision = 1e-3;
struct BlockOfX {
  int length;
  int first;
};
void initialise(int n, double* x, double*);
void writemat(ostream&, double*);
void write(ostream& out, double *x);
void parent(BlockOfX* wholeBlock);
void child(BlockOfX* localBlock);
double x[n],
       y[2][n],
       A[n*n];
double lastnorm, norm;
int proccount, finished, curry;

THREAD_MONITOR controlFlag;

void main(int nargs, char *argv[])
{
  long start_time, stop_time;
  initialise(n,x,A);

  if (nargs>1) nprocs=atoi(argv[1]); else nprocs=5;

  cout << "n = " << n <<", p = " << nprocs << endl;
  writemat(cout, A);

  time(&start_time);

  BlockOfX wholeBlock;
  wholeBlock.length = n;
  wholeBlock.first = 0;
  THREADgo(nprocs,dataSize,parent,&wholeBlock,sizeof(wholeBlock),stackSize,
           priority);
  time(&stop_time);
  cout << "The Eigenvec is:\n";
  write(cout,x);
  cout << "The Eigenvalue is:\n" << lastnorm << endl;
  cout << "Elapsed time: \n" << stop_time-start_time << " sec\n";
  cout.flush();
}

void initialise(int n, double* x, double* A)
{
  for(int i=0; i<n; i++)
  {
    x[i] = 1;
    for(int j=0; j<n; j++)
      A[i*n+j]=((i+2)*(j+6)*5) % 20 -8;
  }
}
void writemat(ostream& out, double *A) {
  for (int i=0; i< n; i++) {
    write(out, &A[i*n]);
  }
}
void write(ostream& out, double *x)
{
  for(int i=0; i<n; i++)
  {
    out.width(12);
    out << x[i];
  }
```

```
  out << endl;
}
void parent(BlockOfX* wholeBlock)
{
  int numThreads = nprocs;
  int startIndex = 0;
  int i;
  controlFlag = THREADmonitorinit(1,NULL);
  proccount=nprocs;
  lastnorm=1;
  curry=0;
  for(i=0; i<n; i++) y[curry][i]=x[i];
  for(i=0; i<numThreads; i++)
  {
    BlockOfX localBlock;
    localBlock.length = (*wholeBlock).length*(i+1)/numThreads - startIndex;
    localBlock.first = startIndex;
    THREADcreate(child,&localBlock,sizeof(localBlock),ATTACHED,stackSize,
                 priority);
    startIndex = startIndex+localBlock.length;
  }
  for(i=0; i<numThreads; i++)
  {
    THREADjoin();
  }
  for(i=0; i<n; i++) x[i]=y[curry][i];
}

double fabs(double x) {
  return x>0?x:-x;
}
void child(BlockOfX* localBlock)
{
  double maximum=0;
  int from=(*localBlock).first,
      to=from+(*localBlock).length,
      i;
  while (1) {

    // calculating the dotproducts and resize the vector
    for(i=from; i<to; i++) {
      double sum=0;
      for(int j=0; j<n; j++) {
         sum += y[curry][j]*A[i*n+j];
      };
      sum /= lastnorm;
      if (fabs(sum)>maximum) maximum=fabs(sum);
      y[1-curry][i]=sum;
    }

    THREADmonitorentry(controlFlag,NULL);

    // update the global norm
    if (norm<maximum)
      norm=maximum;

    proccount--;
    if (! proccount) {       // if this is the last child to updated the norm
      if (fabs(lastnorm-norm)<precision)
        finished=1;
      proccount=nprocs;
      lastnorm=norm;
      norm=0;
      curry=1-curry;
      THREADmonitorsignalandexit(controlFlag,0);
    } else if (proccount == 1) {  // if this is the child before the last one
      THREADmonitorwait(controlFlag,0);
      THREADmonitorexit(controlFlag);
    } else {                 // if at least two childs did not update the norm
      THREADmonitorwait(controlFlag,0);
      THREADmonitorsignalandexit(controlFlag,0);
    }

    if (finished) return;
    maximum=0;

  }
}
```

4.2 PVM – master

```
#include "pvm3.h"
#include <iostream.h>
#include <time.h>
#include <stdlib.h>

#define SLAVENAME "slave"
#define DATATAG     0
#define MATRIXTAG   2
#define RESULTTAG   1
#define MAXPROC    32

void check(int nproc, int numt, int* tids)
{
  int i;

  if (numt < nproc)
  {
    cout << "Trouble spawning slaves. Aborting. Error codes are:\n";
    for (i = numt; i < nproc; i +=1)
    {
      cout << "TID " << i << ' ' << tids[i] << endl;
    }
    for (i = 0; i < numt; i += 1)
    {
      pvm_kill(tids[i]);
    }
    pvm_exit();
    exit(1);
  }
  return;
}
void initialise(int n, float *&x, float *&A)
{
  A = new float[n*n];
  x = new float[n];
  int i,j;
  for (i = 0; i < n; i += 1)
    x[i] = 1;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      A[i*n+j]=((i+5)*(j+10)*3)%20-10;
}
void writevec(int n, float *x) {
  for (int i=0; i<n; i++) {
    cout.width(12);
    cout << x[i];
  }
  cout << endl;
}
void writemat(int n, float *A) {
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      cout.width(12);
      cout << A[i*n+j];
    }
    cout << endl;
  }
}
float fabs(float x) {
  return x<0?-x:x;
}
int main(int nargs, char *argv[])
{
  int mytid;                  /* my task id */
  int tids[MAXPROC];          /* slave task ids */
  int bufids[MAXPROC];        /* message buffers */
  int n, nproc, numt, i, info, bufid, start, end;
  float *x, *A, loc_sum, sum,
        locnorm, lastnorm, norm;
  struct pvmhostinfo *hostp[MAXPROC];
  long int start_time, stop_time;

  mytid = pvm_mytid();
  nproc = 5;
  if (nargs>1) nproc=atoi(argv[1]);

  n = 4;
  initialise(n, x, A);
```

```
  cout << "n = " << n << ", p = " << nproc << endl;
  writemat(n,A);

  time(&start_time);

  numt = pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
  check(nproc, numt, tids);

  for (i = 0; i < nproc; i += 1)
  {
    bufids[i] = pvm_mkbuf(PvmDataDefault);
  }

  end = 0;
  for (i = 0; i < nproc; i += 1)
  {
    start = end;
    end = ((i+1)*n)/nproc;
    bufid = pvm_initsend(PvmDataDefault);
    info = pvm_pkint(&n, 1, 1);
    info = pvm_pkint(&start, 1, 1);
    info = pvm_pkint(&end,1 , 1);
    info = pvm_pkfloat(&A[n*start], n*(end-start), 1);
    info = pvm_send(tids[i], MATRIXTAG);
  }

  norm=1.0;
  lastnorm=0.0;
  while (fabs(lastnorm-norm)>0.001) {
    for (i = 0; i < nproc; i += 1)
      {
        bufid = pvm_initsend(PvmDataDefault);
        info = pvm_pkfloat(&norm, 1, 1);
        info = pvm_pkfloat(x, n, 1);
        info = pvm_send(tids[i], DATATAG);
      }
    lastnorm=norm;
    norm = 0.0;
    for (i = 0; i < nproc; i += 1)
      {
        info = pvm_recv(tids[i], RESULTTAG);
        info = pvm_upkint(&start, 1, 1);
        info = pvm_upkint(&end, 1, 1);
        info = pvm_upkfloat(&x[start], end-start, 1);
        info = pvm_upkfloat(&locnorm, 1, 1);
        if (locnorm>norm) norm=locnorm;
      }
  }
  time(&stop_time);
  cout << "Eigenvector is:\n";
  writevec(n, x);
  cout << "Eigenvalue is:\n" << norm << endl;
  cout << "Elapsed time:\n" << stop_time-start_time << " sec\n" << flush;
  for (i = 0; i < numt; i += 1) {
    pvm_kill(tids[i]);
  }

  info = pvm_exit();
  return 0;
}
```

4.3 PVM – slave

```
#include "pvm3.h"
#include <iostream.h>

#define DATATAG     0
#define MATRIXTAG   2
#define RESULTTAG   1

float sdot(int, float*, float*);
float fabs(float x) {
  return x<0?-x:x;
}
int main()
{
  int master, mytid, n, start, end, info, bufid, i;
  float *x, *A, *y, locnorm, norm;
```

```
  mytid = pvm_mytid();
  master = pvm_parent();

  info = pvm_recv(master, MATRIXTAG);
  info = pvm_upkint(&n, 1, 1);
  info = pvm_upkint(&start, 1, 1);
  info = pvm_upkint(&end, 1, 1);
  A = new float[n*(end-start)];
  x = new float[n];
  y = new float[end-start];
  info = pvm_upkfloat(A, n*(end-start), 1);

  while (1) {
    info = pvm_recv(master, DATATAG);
    info = pvm_upkfloat(&norm, 1, 1);
    info = pvm_upkfloat(x, n, 1);

    locnorm=0;
    for (i=0; i<end-start; i++){
      y[i]=sdot(n, &A[i*n], x)/norm;
      if (fabs(y[i]) > locnorm)
        locnorm=fabs(y[i]);
    }

    bufid = pvm_initsend(PvmDataDefault);
    info = pvm_pkint(&start, 1, 1);
    info = pvm_pkint(&end, 1, 1);
    info = pvm_pkfloat(y, end-start, 1);
    info = pvm_pkfloat(&locnorm, 1, 1);
    info = pvm_send(master, RESULTTAG);
  }
  info = pvm_exit();
  return 0;
}

float sdot(int n, float* x, float* y )
{
  int i;
  float sum = 0.0;

  for (i = 0; i < n; i += 1)
  {
    sum += x[i] * y[i];
  }
  return sum;
}
```