

# Multivariate Minimization Using Grid Computing

Kandle Kulish      Jerry Perez      Phil Smith  
High Performance Computing Center  
Texas Tech University

## Abstract

In November 2002, Texas Tech implemented a production grid powered by Avaki software. This paper addresses a new, simple approach to parallel programming which requires little or no actual knowledge of parallel programming structures, such as MPI, enabling even the novice user to take advantage of this resource. We describe and test this feature of grid computing called multiple input files. To demonstrate and test this feature, we will solve a toy multivariate global minimization problem. We would like to solve  $\min\{f(x) : a_i \leq x_i \leq b_i\}$ , where  $a$  and  $b$  are the vectors which define the boundary of our domain. Using the multiple input file feature we subdivide the region, and each subregion is solved on a different processor on the grid yielding linear speed ups and more accuracy for the same amount of compute time.

## 1 INTRODUCTION

As technology advances, so does our scientific curiosities and capabilities. We are now at the point that even with our fastest supercomputers some calculations would take months or years to compute. To help speed up these computations, parallel computing developed, which could partition the workload of a program onto multiple processors. But with the rising costs to purchase and maintain these multiprocessor supercomputers, a more efficient solution had to be found. One such solution is the Beowulf cluster, which links inexpensive Linux machines together and supports parallel computing. But what about the Windows personal computers in offices and labs that have unused compute cycles? Thus the concept of grid computing emerged to link together not only Windows machines, but Beowulf clusters and supercomputers as well. Grids [1] link all your resources together into one giant heterogeneous compute and data grid. Texas Tech implemented a production grid powered by Avaki software in November 2002 called TechGrid. In this paper, we will discuss a feature of grid computing : multiple input files. This feature allows novice users to access significant computing power with little or no knowledge of parallel computing methodologies. Basically, this is a way of forcing parallelism without having to use any parallel coding structures such as MPI. We will illustrate this feature via a toy global minimization problem. The actual technique of minimization currently used in this example is rather simplistic since we are focusing mainly on the new grid feature. By using the multiple input files to subdivide the domain into regions, a more accurate global minimum is probable. Further sophistication in the minimization algorithm is currently a work in progress.

## 2 MULTIVARIATE MINIMIZER

### 2.1 The Problem

One problem of interest in the mathematical world is the minimization of functions. Functional minimization is needed in many real world applications such as solving nonlinear

ordinary and partial differential equations, financial modeling, plant optimization, and resource allocation. Our goal is to write a computer program which will minimize a function over an n-dimensional domain. For simplicity we will define our n-dimensional domain by two vectors of length  $n$ . The first,  $a$ , will define the lower left corner of our domain and the second,  $b$  will define the upper right corner of our domain. For the purpose of testing, we chose a function for which we know the true minimum. For this model problem, we define the functional value to be the sum of the squares of the  $x_i$ 's over all  $i$ , i.e.

$$f(x_1, x_2, \dots, x_n) = \sum_{i=1}^n (x_i)^2. \tag{1}$$

Thus, for a function of this nature, the true minimum would occur at the lower left corner defined by vector  $a$ , when considering  $a \geq 0$ . So the problem can be stated as

$$\min\{f(x) : a_i \leq x_i \leq b_i, i = 1, \dots, n\}. \tag{2}$$

## 2.2 The Coding Approach

For a complete copy of all programs used in this experiment, see the appendix. The main program, *mini\_no\_p.c* begins by reading in from an input file the dimension of the domain, the number of iterations, and the vectors  $a$  and  $b$  which will define the lower and upper limits of the domain over which our function will be minimized. After echoing the input data into the output file, the function *mini\_me()* is called and the approximate minimum and where it occurs is written to the output file. The function *mini\_me()* is a prototype of all function minimizers. It trivially uses function values to produce an approximate minimum by generating a random vector within the domain and testing the functional value of the vector. If the functional value being tested is less than the current stored minimum, it copies the minimum and stores the vector where this minimum occurs. The program will repeat this process of generating a random vector and testing as many times as was specified in the input file.

# 3 GRID COMPUTING

## 3.1 What is a Grid

A Grid is a dynamic network of computing resources that work together as a single, uniform operating environment. The term grid computing originated in the early 1990's as a metaphor for making computer power as easy to access as an electric power grid. Grids can span locations and administrative domains, and can flexibly support dynamically changing organizations and computing requirements. The grid distributes compute jobs among compute nodes within the grid using middleware to facilitate distributed computing. TechGrid, which uses Avaki grid middleware, is the union of networked Texas Tech University desktop Windows lab machines, office machines, and Linux Beowulf clusters.

## 3.2 Why We Care

Grid computing is an amazing alternative to classical distributed computing. Rather than running parallel or serial jobs on a lone desktop pc, a Beowulf cluster, or even a supercom-

puter, grid computing unifies all your available resources which allows for more options. You can even run your multiprocessor jobs on an entire lab of windows machines completely in the background, taking advantage of unused compute cycles. In the case of this multivariate minimizer serial example, we are able to use the many processors at our fingertips to our advantage. Instead of iterating and finding minima and vectors across the entire domain on a single processor, we used the program *create\_em.c* to generate multiple input files which partition the domain into regions. Each one of these regions may then be farmed out to different processors for local minimization. In doing this, you are dividing and conquering the problem, and a global minimum closer to the actual minimum is more likely. Also, our serial code now becomes embarrassingly parallel without having to use any parallelized coding structures such as MPI.

The Grid handles all the coding changes for us. We can still use the same serial code which asks for input file *input.txt* and writes to *output.txt*. We subdivide our domain into  $N$  partitions, making  $N$  input files :

*input1.txt, input2.txt, ... , inputN.txt.*

A simple, one line command is then executed to run the job on the grid. The grid first identifies available compute resources. Next, the grid determines that you have  $N$  input files which match the requirements for the program and executes the following loop  $N$  times :

1. Locate an available node.
2. Move the executable and input file *input\*.txt* onto the available node.
3. Rename the input file to *input.txt*
4. Run the executable, which will compute the minimum and write the results to the file *output.txt*.
5. Rename *output.txt* to *output\*.txt* to match the pattern of the original input file.
6. Remove the temporary files such as the executable, *input.txt*, *output.txt* (clean up).

## 4 RUNNING THE CODE

### 4.1 Example Problem

For the purpose of comparison, we chose to test our code both on a single pc and as a grid application using the pc's connected to the grid to see which could find a better minimization. On both, we chose the number of dimensions to be 3 on a domain defined by  $a = (1.0, 1.0, 1.0)$  and  $b = (7.0, 7.0, 7.0)$ , with the number of random vectors generated and tested to be 3000. Since the grid will support multiple input files, we again chose to subdivide our domain into thirds and thus read in the following as our  $a$ 's and  $b$ 's to define the three subdomains :

$$\begin{array}{ll}
 a_1 = (1.0, 1.0, 1.0) & b_1 = (7.0, 7.0, 3.0) \\
 a_2 = (1.0, 1.0, 3.0) & b_2 = (7.0, 7.0, 5.0) \\
 a_3 = (1.0, 1.0, 5.0) & b_3 = (7.0, 7.0, 7.0)
 \end{array}$$

For this data then, our input files were as follows:

input1.txt looked like the following :

```
3
3000
1.0 1.0 1.0
7.0 7.0 3.0
```

input2.txt looked like the following :

```
3
3000
1.0 1.0 3.0
7.0 7.0 5.0
```

input3.txt looked like the following :

```
3
3000
1.0 1.0 5.0
7.0 7.0 7.0
```

## 4.2 Results

Due to the nature of the function, the true minimum for this example would occur at (1.0, 1.0, 1.0), yielding 3.0 as the minimum. After running on a pc for the described example problem from (1) and (2), we found the minimum of the test function to be 4.114095 having occurred at the vector (1.359719, 1.001212, 1.123759). The output files from the grid execution were as follows :

output1.txt looked like the following :

```
3
3000
1.0 1.0 1.0
7.0 7.0 3.0
```

```
3.935468
1.359719 1.001212 1.041253
```

output2.txt looked like the following :

```
3
3000
1.0 1.0 3.0
7.0 7.0 5.0
```

```
12.100481
1.359719 1.001212 3.041253
```

output3.txt looked like the following :

```
3
3000
```

1.0 1.0 5.0  
7.0 7.0 7.0

28.265493  
1.359719 1.001212 5.041253

After computing on the grid, the three local minima found were 3.935468, 12.100481, and 28.265493 having occurred at the vectors (1.359719, 1.001212, 1.041253), (1.359719, 1.001212, 3.041253), and (1.359719, 1.001212, 5.041253) respectively. So, the approximate global minimum found using the grid was 3.935468. In this case, there was only a minor improvement in the global minimum. Of course, this is due to the simple nature of the example and not a limitation of the method.

## 5 CONCLUSIONS AND FUTURE RESEARCH

By running this application on our grid and taking advantage of its multifile capabilities, we were able to find a minimum which was much closer to the true minimum of our function. We would like to continue our exploration of functional minimization in several ways. We would like to construct a grid application which would read through all the output files generated, by searching the subdomains and calculate the global minimum found over all of the domain. This would allow us to then implement an adaptive search technique which would take the minimum found and draw a "box" around it and further subdivide this domain and run the multifile search again to look for a better minimum. Additionally, we would like to explore implementing more sophisticated techniques to search for our minimum such as simulated annealing or the Nelder Mead Downhill Simplex Method [2].

More importantly there are many computational areas that have "embarrassingly" parallel applications that fit the multifile paradigm. Examples of this would be : solution of stochastic differential equations, factorization of large integers, numerical integration, and genomic computing (BLAST).

## References

- [1] Foster I. , and Kesselman C., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, San Francisco, (1999).
- [2] Press W. H. , Tenkolsky S. A. , Vetterling W. T. , and Flannery B. P. , *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, Cambridge, (1992).
- [3] Salamon P. , Sibani P. , and Frost R. , *Facts, Conjectures, and Improvements for Simulating Annealing*, SIAM, Philadelphia, (2002).

## 6 Appendix

### 6.1 *create\_em.c*

```
// Kandle Kulish
// 4-14-2003
// create_em.c
// The purpose of this program is to generate the
// input files to use with the program mini_no_p.c
// As it is written, three files will currently be
// created as is defined by numfiles. I have
// the program written in such a way that it
// is dividing the domain into subregions in the
// last dimension.

#define numfiles 3
#define dimension 3
#define iterations 30000

#include <stdio.h>
#include <string.h>

main()
{
    char filename[10];
    int i,j;
    FILE *fp;
    double a[dimension] = {1.0, 1.0, 1.0};
    double b[dimension] = {7.0, 7.0, 7.0};
    double a_new[dimension];
    double b_new[dimension];

    double jump = (b[dimension-1] - a[dimension-1])/numfiles;

    for (i = 0; i < dimension; i++)
    {
        a_new[i] = a[i];
        b_new[i] = b[i];
    }
    b_new[dimension-1] = a_new[dimension-1] + jump;

    for (i = 1; i <= numfiles; i++)
    {
        sprintf(filename,"input%d.txt",i);
        fp = fopen(filename,"w");
        fprintf(fp,"%d %d\n",dimension,iterations);
    }
}
```

```

    for (j=0; j<dimension; j++)
        fprintf(fp,"%f ",a_new[j]);
    fprintf(fp,"\n");
    for (j=0; j<dimension; j++)
        fprintf(fp,"%f ",b_new[j]);
    fclose(fp);
    a_new[dimension-1] = a_new[dimension-1] + jump;
    b_new[dimension-1] = b_new[dimension-1] + jump;
}
}

```

## 6.2 *mini\_no\_p.c*

```

// Kandle Kulish
// Multivariate Minimizer non parallelized version
// 4-9-2003h
// mini_no_p.c
// The purpose of this program is to find the minimum of a function defined
// in the subroutine funky. It solves for the minimum by generating a random
// vector within the boundary box of [a,b] and testing the functional value
// of that vector. If the functional value of the generated vector is less
// than the current minimum, it stores the new minimum and keeps track of
// the vector that produced it. The program will generate M such vectors
// for testing.

// -----include files-----

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "randomlib.h"
#include "randomlib.c"

// -----function prototypes-----

double funky (double *x, int n);

double mini_me (double a[],double b[],double mini_where[],int
n,int m);

double (*p)(double *x,int n);          // pointer declaration

// -----main program-----

main()

```

```

{ // BEGIN main
  // The main program defines the boundary within which the function is to be
  // minimized with a, b. It then calls the subroutine mini_me to minimize the
  // function and prints out the minimum and where it occurred.

  int i,N,M; // looping variable
  double temp;

  double *a; // lower left of boundary box
  double *b; // upper right of boundary box
  double *mini_where;

  FILE *fp1, *fp2;

  fp1 = fopen("input.txt","r");
  fp2 = fopen("output.txt","w");

  fscanf(fp1,"%d",&N);
  fscanf(fp1,"%d",&M);

  a = (double*) calloc(N,sizeof(double));
  b = (double*) calloc(N,sizeof(double));
  mini_where = (double*) calloc(N,sizeof(double));

  for (i = 0; i < N; i++)
  {
    fscanf(fp1,"%lf",&temp);
    a[i] = temp;
  }
  for (i = 0; i < N; i++)
  {
    fscanf(fp1,"%lf",&temp);
    b[i] = temp;
  }
  fprintf(fp2,"%d\n%d\n",N,M);
  for (i = 0; i < N; i++)
    fprintf(fp2,"%f ",a[i]);
  fprintf(fp2,"\n");
  for (i = 0; i < N; i++)
    fprintf(fp2,"%f ",b[i]);

  fprintf(fp2,"\n\n\n%f\n\n",mini_me(a,b,mini_where,N,M));
  for (i = 0; i < N; i++)
    fprintf (fp2,"%f ",mini_where[i]);

  fclose(fp1);

```

```

    fclose(fp2);

} // END main

// -----

double funky (double *x, int n)
{ // BEGIN funky
    // INPUT : a pointer to vector x (of length n) to find the functional value of,
    //          and an integer n to store the length of the vector
    // OUTPUT: the functional value of the input vector
    //
    // This subroutine defines another function we want to minimize. In this case,
    // we define the functional value to be the sum of the squares of the x[i]'s
    // over all i (length of vector). The functional value is returned once computed.

    double sum = 0.0;
    int i;

    for (i = 0; i < n; i++)
        sum = sum + x[i]*x[i];

    return sum;
} // END funky

// -----

double mini_me(double a[],double b[],double mini_where[],int n,
int m)

{ // BEGIN mini_me
    // INPUT : vectors a,b of doubles to store the bottom left corner and upper
    //          right corner, respectively, of the boundary box to find the minimum
    //          within, a vector mini_where to store the location the minimum occurred,
    //          an integer n to store the length of the vector, and an integer m to
    //          control how many random vectors will be tested in search of a minimum.
    // OUTPUT: the minimum functional value
    //
    // This subroutine generates a random vector m times and tests the functional
    // value of each random vector generated. If the functional value being tested
    // is less than the current stored minimum, it copies the minimum and stores the
    // vector where this minimum occurs. The minimum is returned when finished.

    double mini_what; // initialize minimum
    double *x; // temporary vector storage

```

```

double rmin, rmax;
int i,j,k;

x = (double*) calloc(n,sizeof(double));
p = funky; // points to funky

for (j=0; j < n; j++) // initialize mini_what and where
{
    rmin = a[j];
    rmax = b[j];
    mini_where[j] = RandomDouble(rmin,rmax);
}
mini_what = p(mini_where,n);

for (i = 0; i < m; i++)
{
    for (j = 0; j < n; j++) // generates random vector to test
    {
        rmin = a[j]; // lower bound for random number
        rmax = b[j]; // upper bound for random number

        x[j] = RandomDouble(rmin,rmax); // calls random number generator prog
    }

    if (p(x,n) < mini_what) // tests if min is less than current min
    {
        mini_what = p(x,n); // if so, copies into min and stores the
        for (k = 0; k < n; k++) // vector where this minimum occurs.
            mini_where[k] = x[k];
    }
}
return mini_what;
} // END mini_me

```