WORKING PAPER 96/13

A Social Science Benchmark (SSB/1) Code for Serial, Vector and Parallel Supercomputers

Stan Openshaw
and Joanna Schmidt

Published July 1996

For further copies contact the Working Paper Secretary,
School of Geography, University of Leeds, Leeds, LS2 9JT
Telephone 0113 233 3300

Abstract

This paper describes a social science benchmark suitable for evaluating the performance of serial, vector and parallel supercomputers. The benchmark suite implements a spatial interaction model that represents a typical class of computational problems performed by social scientists.

The computations in the benchmark are highly vectorisable and highly parallelisable. The benchmark allows for workload adjustment and permits scaling experiments with datasets of virtually any size that may be considered important in the future. The code is designed to be easily portable and run on hardware ranging from PCs to the latest and largest superco,puters. The URL for the World Wide Web page for the benchmark is:

`http://www.leeds.ac.uk/ucs/projects/benchmarks/index.html`

1. Introduction

The main objective of benchmarking is to measure a performance characteristic of a computer system. Therefore, the aim of benchmarking computer hardware is to evaluate how well, how fast and how efficiently handles a software application that can in some senses be viewed as being representative of what a class of users may wish to run on the hardware. The importance of benchmarking increases more or less in proportion to the cost of the hardware, as indeed does the complexity of the benchmarking task. The Social Sciences are an important class of computer users and until now have not had a benchmark code that can be used to measure the performance of high performance computer systems on any of their types of application. Indeed, the procurement process that has been responsible for purchasing all of the UK's national supercomputing facilities, has not used any benchmark that can claim to be even slightly representative of what geographers and other social scientists might well wish to run on the hardware. Equally important is the realisation that the lack of benchmarking also means that social scientists may well be undervaluing the very significant developments going on in the High Performance Computing (HPC) world and their potential benefits for solving social science computational problems, because the prospective users simply do not know what the usual benchmark measures mean in terms of applications that bear even a slight resemblance to what they do. For example, how much slower or faster is a Cray J90 compared with a 200MHz Pentium PC or a Cray T3D with 512 processors and thus what type of problems may be computed on the available hardware.

Computer manufacturers often measure the performance of their hardware in MIPS (millions of instructions per second) or Mflops (millions of floating point instructions per second). These peak ratings do not always adequately reflect the performance of a real-life application on a computer system as machine performance may vary from program to program depending on a type of operations performed by the code. It is important therefore to evaluate performance using a real-life application that portrays the behaviour of a typical program run on a machine.

As a result a number of standard benchmarks (programs) have been produced and are widely used to measure the performance of hardware ranging from PCs to supercomputers. In practice these *benchmark codes* might be a specially modified version of a real application or a specially designed suite of *typical* operations thought likely to be performed. Examples of synthetic codes are the Dhrystone and the Whetstone benchmarks, that have been applied to a large number of computers. Perhaps, the most popular has become Dongarra's Linpack suite comprising a set of linear algebra subroutines used for solving dense systems of linear equations (see Dongarra, 1995). In 1988, the Standard Performance Evaluation Co-operation (SPEC) was formed and a new SPECmark89 suite was soon released, followed by the SDM suite in 1991, the SPECmark92 in 1992, and recently by the SPECmark95. Other popular application benchmarks are the Livermore Loops and Perfect Club suite, that covers fluid flow, signal processing, engineering design and physical and chemical modelling areas (see Lewis and El-Rewini, 1992). An example of the benchmark providing distributed memory message passing code is the GENESIS code. An overview of these and other benchmarks may be found in the publication (Blocklehurst, 1991) issued by the National Physical Laboratory (NPL), which is a member of an ESPRIT project on Performance Evaluation of Parallel Systems (PEPS). All these benchmarks have certain drawbacks and some of them have become obsolete with the development of computer technology. More importantly none of them adequately reflects the computational nature of any typical social science problems requiring use of HPC.

The objective of this paper is to define a self-contained, universal, benchmark code that is thought to be broadly representative of a particular class of social science HPC. The benchmark code is based on a singly and doubly constrained spatial interaction model (Wilson, 1974) and is designed to be simple to implement. This model is still of considerable value in various application areas, it has been ported onto various parallel and vector machines and it is believed to be broadly representative of many (but not all) kinds of computer processing that social scientists do or many wish to do on HPC hardware. Indeed, it is one of the earliest examples of HPC being performed by geographers; see for example, Openshaw (1987). The proposed benchmark is based on a mathematical model that has the advantage of being naturally data parallel. As a result it has been written in such a form that it can be efficiently executed on serial, vector and parallel machines. Furthermore, it is based on a real world application but has been made scalable so that a wide range of machine types and sizes can be handled.

2. The Spatial Interaction Model Benchmark

There is a family of spatial interaction models (SIMs) that describe the flow of goods or people between geographic areas which act as origins and destination zones. Models of this type are widely used in retail site evaluation, in transportation planning, and location optimisation (Birkin et al, 1996). Two versions of this model are of interest as a benchmark.

2.1. Origin Constrained Model

Model (1) is an origin (singly) constrained model which can be specified as:

$$T_{ij} = D_j O_i A_i \exp(-\beta C_{ij}) C_{ij}^{\alpha} \qquad (1)$$

with:

$$A_i = 1 / \sum_{j=1}^{M} D_j \exp(-\beta C_{ij}) C_{ij}^{\alpha} \qquad (2)$$

to ensure that:

$$\sum_{j=1}^{M} T_{ij} = O_i \qquad i=1,\ldots,N \qquad (3)$$

where:

T_{ij} is the number of trips (flows) between zone i and zone j

O_i is the number of trips starting in zone (origin) i

D_j is the size of zone j

C_{ij} is the distance between origin i and destination j

α, β are parameters

N is the number of origins

M is the number of destinations

The (`exp (-βCij )Cij`$^{\alpha}$) expression represents the trip deterring effect of distance or cost. The matrices `T` and `C` contain `N` (rows) origins and `M` columns (destinations). The amount of computations performed depends on the size of `N` and `M`. For example in equations (1) and (2) there are

`N x M x 8` multiplications

`N x M x 2` calls of the `exp` maths function and exponentiation

`N x M x 2` calls to the `power` function

`N x M` additions

`N` divisions

Additionally, the model would normally be run as a subroutine that could well be called a large number of times; for example, in a spatial optimisation problem such as that of Birkin et al (1995). The modelling process also requires the computation of an error measure such as the sum of squares:

$$F = \sum_{i=1}^{N} \sum_{j=1}^{M} \left(T_{ij}^{predicted} - T_{ij}^{observed} \right)^2 \qquad (4)$$

which involves a further

`N x M` subtractions

`N x M` squares (or multiplications)

`N x M` additions

The count of floating point operations performed by the code can be used for determining the Mflops delivered by the hardware as viewed from this particular application.

2.2. A Doubly Constrained Model

Model (2) is a doubly constrained model which is specified as:

$$T_{ij} = O_i D_j A_i B_j \quad \exp(-\beta C_{ij}) C_{ij}^{\alpha} \qquad (5)$$

where:

$$A_i = 1 / \sum_{j=1}^{M} B_j D_j \exp(-\beta C_{ij}) \, C_{ij}^{\alpha} \tag{6}$$

$$B_j = 1 / \sum_{i=1}^{N} O_i A_i \quad \exp(-\beta C_{ij}) C_{ij}^{\alpha} \tag{7}$$

where T_{ij}, O_i, D_j, C_{ij}, N , M have the same meaning as described previously.

This model is constrained at both ends: viz.

$$\sum_{j=1}^{M} T_{ij} = O_i \qquad \text{i=1,...,N} \tag{8}$$

and.

$$\sum_{i=1}^{N} T_{ij} = D_j \qquad \text{j=1,...,M} \tag{9}$$

where T_{ij}, O_i, D_j, C_{ij}, N , M , α, β have the same meaning as described previously.

Versions of this model are used in modelling traffic flows in transportation planning. The doubly constrained model (equations 4,5,6) is more complex than the singly constrained model, because the A_i and B_j terms are functions of each other and have to be estimated iteratively. To avoid variable amounts of the computational work being performed here the number of iterations performed is fixed at 20.

In both models the quality (i.e. spatial resolution) of the results depends on both N and M and the number of times the model can be re-run for different O_i or D_j or β and α values in an acceptable (i.e. fixed) length of time. Clearly as N and M become large so the computational load increases accordingly. Small N and M values (i.e. 100) can be run on a PC, large values (i.e. 10,000) need a Cray T3D, and possible maximum values (i.e. 1.6 million) may need the next generation or two of highly parallel supercomputers.

3. Characteristic Features of the Spatial Interaction Model Benchmark

The benchmark has a number of key features making it particularly suitable for assessing the performance of serial, vector and parallel computers:

(1) The code contains only a few hundred lines of portable FORTRAN therefore it is easy to understand and port to a new platform. Porting a large code is not a trivial

task and requires a major effort by vendors or developers. A useful benchmark requires the minimum of vendor time and effort to apply it.

(2) The benchmark has a built-in data generator so there is no need to store any data on disk or perform large amounts of input and output during the benchmarking. This also avoids the problem of shipping large volumes of data around and yet permits benchmark runs to be performed on realistic and variable sizes of datasets. The self contained nature of the benchmark code is another important ease of use feature.

(3) The benchmark code permits a wide range of dataset sizes to be investigated providing a platform for various numerical experiments and allows scaling experiments with datasets of virtually any size that may be considered future important.

(4) The executable time and memory size of the benchmark are easily adjustable (alter the problem size i.e. `N, M` values). A standard set of ten `(N, M)` values is suggested that reflect different sizes of real world application and to permit scalability and performance measurement experiments on comparable problems on different hardware.

(5) The ratio of computation to memory references in the benchmark is typical of social science problems requiring the use of high performance computers. Much social science computing involves a high ratio of memory accessing to computation. In many statistical models, and also here, there is a memory access for each floating point operation performed.

(6) The performance indicators can be readily interpreted because it is easy to establish a model of the computational load being generated by the benchmark.

(7) The load balancing of individual processors on parallel machines is readily achievable because of the parallel nature of the model.

(8) The benchmark has a built-in results verifier that checks against the reference values whether the numerical results are acceptable for a standard set of the benchmarks.

(9) The benchmark code is available in the public domain and can be easily downloaded.

(10) The benchmark manipulates the sparse data structures that are common to many real-life applications.

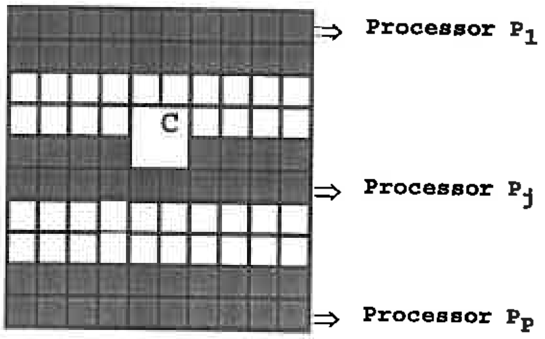
A key issue of machine performance concerns the ratio of arithmetic operations to memory references. The drawbacks of many benchmarks are that there are very few memory access operations performed, however, in most social science applications there is typically little computation executed in comparison to the number of memory references, e.g. dot products, inner products on large array sizes. Also, large numbers of mathematical function calls (i.e. LOG, EXP) are made, but the effect of these often gets lost in counts of flops or, worse still, are simply excluded altogether. Moreover, these mathematical functions are not always highly optimised and can absorb a large fraction of total compute times. In addition to measuring flops the benchmark also has a reasonably representative level of the non floating point register load as it uses sparse matrix methods to store the flow data. The need to perform large amounts of integer arithmetic is often overlooked in benchmarks and in hardware designed solely to optimise pure number crunching throughput on data stored in a highly optimised cache. Social science HPC tends to be much more data intensive than many other areas of science.

4. Data Used for Benchmarking

The benchmark uses artificially generated data, which preserve the sparsity pattern and general structure of real flow data. A data generator has been constructed that permits the generation of virtually any size of dataset. This approach offers some benefits; it allows investigations of different problem sizes, it avoids the need for obtaining permission to access the statistics and thus circumvents potential legal data copyright problems, and it can also simulate flow data of much greater volumes than the current largest available datasets. In the UK the largest public domain flow table has 10,764 rows and columns, but the maximum flow datasets that exist could well have between 1.7 and 28 million rows and columns (e.g. telephone flows between houses). The data generator is designed to always create the same data, regardless of hardware or floating point precision, for the same N and M values.

5. SIM Parallelisation and Data Partitioning Strategies

The basic models described in equations 1-2, 5-7 are easily expressed as vector or serial operations. Parallelisation is a little more difficult. The models can be parallelised at two levels: assign each model to a separate processor (whole model parallelism) and use multiple processors to compute a single model (within model parallelism). The former is trivial but problem sizes may be memory limited and it also restricts parallelism to these applications that require multiple simultaneous model calls (e.g. bootstrapping and optimisation). The latter form of more finely grained parallelism needs further elaboration as it is not straightforward. The SIM, like many other social science applications, is memory-bounded.* The implementation of this type of problem and in particular the data distribution strategy may affect the size of the problem that can be computed on a parallel system. This is important because current trends in advanced HPC architecture are towards scalable parallel systems comprising a number of individual processing units with a local memory (e.g. Cray T3D) connected by a high bandwidth interconnect. The local memory per processing node is limited and usually not very large, however, when used collectively it will permit the solution of a very large model. The challenge is to implement a data distribution strategy that will take advantage of this type of architecture. In the origin constrained model (SC), each processor generates in parallel a block of rows a of the $\{C_{ij}\}$ matrix, all elements of the $\{O_i\}$ and $\{D_j\}$ matrices. Each processor computes a group of rows of the trip matrix. The rows are distributed in blocks (see Figure 1), e.g. processor 1 computes rows `1` to `N/P`, processor 2 computes rows `N/P+1` to `2*N/P`, where `P` is equal to the number of available processors.



Partitioning of Cost Matrix and Work Distribution for the SC model

The same strategy is used for the doubly constrained (DC) model except that the $\{c_{ij}\}$ values also need to be distributed by columns. Each processor generates a block of rows that are divided into smaller chunks containing blocks of columns and then scattered among other processors.

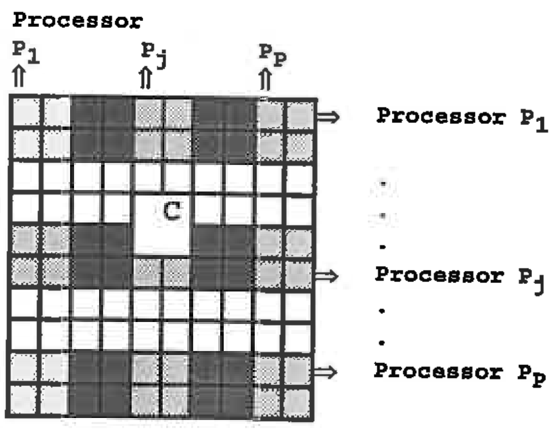
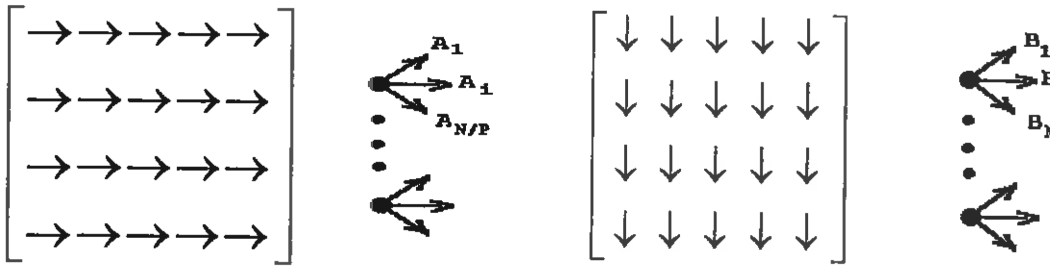


Figure 2: Partitioning of Cost Matrix and Work Distribution for the DC model

This process, which is performed in parallel, ensures that each processor has access to a block of rows and a block of columns of the cost matrix (see Figure 2). A close look at the formulas (6) and (7) reveals that each A_i is dependent on all the B_j values, and vice versa. This implies that A_i and B_j must be solved iteratively. Each processor computes a fixed number of rows, and broadcasts the obtained values to all the other processors. Then, each processor computes a block of columns and communicates the calculated values to the remaining processors. This is done repetitively for a certain number of iterations. The parallel algorithm first sweeps forward across the rows computing the A_i values and communicating them to other processors and then sweeps across the columns computing the B_j values and communicating them to all other processors. This process is repeated until a fixed number of iterations are performed (see Figure 3).



Step K
A_i values are computed

Step K+1
A_i values are broadcasted

Step K+2
B_j values are computed

Step K+3
B_j values are broadcasted

Figure 3: Computation of A_i and B_j by Parallel Algorithm for the DC model

Finally, after A_i and B_j are computed, a matrix $\{T_{ij}^{predicted}\}$ is calculated in parallel. Each processor computes a fixed number of rows of the trip matrix using the formula (5).

6. Implementation of the Benchmark

The parallel benchmark implements the single program multiple data (SPMD) approach. This means that the same code is run on all parallel processing nodes. The parallel programming paradigm used here is explicit message passing. Both spatial interaction models are implemented as one program written in Fortran 77. The source code consists of the main program and a set of routines. The parallel routines use the Message Passing Interface (MPI), a current standard for process communication. An MPI version of the benchmark can be run on parallel systems and concurrently on a cluster of workstations supporting the MPI library.
A serial (uniprocessor) version of the benchmark is also written in Fortran 77 and is supplied as a separate program. This code may be run on PCs, Unix uniprocessor workstations and vector supercomputers.

The benchmark also incorporates a results verifier (see paragraph 9) for some of the tests from the standard set. The benchmark comes together with the pre-processor that computes the approximate storage required for arrays in the program specified by problem size (N, M) and P number of processors.

Finally, there are two implementations of the code supplied for both versions of the code, parallel and serial. One implementation of the code applies a single precision for floating point arithmetic with the exception of all sum reductions that are always performed using double precision variables. This code is suitable for running on hardware implementing 32-bit precision for default floating point arithmetic. A second implementation applies a single precision for all real variables throughout the code and may be run on hardware defaulting to 64-bit precision for floating point arithmetic.

7. Benchmark Types

Two types of benchmarks are proposed:

(1) an *as-is code* run in which no modifications are permitted to the original source code. No additional statements or even comments may be inserted in the code. The only exception here is the `simtime` function in the serial code that needs to be replaced by a suitable time measuring routine on a serial system that is being benchmarked.

and

(2) an *optimal code* that contains the original code modified so as to allow tuning of the code to the available architecture. All modifications to the code should be carefully recorded.

Both types of benchmarking should provide a comparison of the *as-is* and *optimal-code* performance.

The benchmarking code may be used for:

(1) measuring hardware flop rate as viewed from the spatial interaction modelling perspective, thereby allowing comparisons to be made between different types of hardware with different architectures and number of processors;

(2) measuring a throughput rate of a computer system by running a large fixed workload on a system and measuring the performance of the system over some time; for example, number of model evaluations per hour;

and

(3) assessment of a scalability of a parallel system by executing a set of jobs for a different numbers of processors and plotting either flops or throughput.

Table 1 (see Appendix 1) contains a suggested set of standard benchmark data set sizes that may be performed. The smallest data set sizes have to be capable of being run on a PC. A standard set of fixed problem sizes is important for the comparison of performance of different serial, vector, and parallel architectures for this class of problems. In addition, it is also possible for users to define their own benchmarks, for example using rectangular flow tables in order to simulate the effects of reducing the size of the parallel regions.

8. Compiling and Running the Benchmark

The benchmark may be executed in a different way on different computational architecture depending on the type of system being evaluated.

The following benchmarking runs are suggested for parallel hardware:

a) a single processor job executing a parallel code that is performed on a dedicated processor (see Table 1 in Appendix A for a description of a job)

b) a parallel job executing concurrently on `P` dedicated processors. We suggest that the `P` parameter should equal the maximum number of processors available on a system.

For serial and vector architecture the following job may be executed:

a) a single processor job executing a serial code that is individually performed (see Table 1 in Appendix A for a description of a job)

The memory requirements of the executable code depend on the size of the dataset used for benchmarking. The program is coded in Fortran 77 which doesn't support allocatable (dynamic) arrays. Before the benchmark code is compiled the parameters `N`, `M`, `P`, `ISC` and `IDC` need to be specified, where `N` is the number of origins, `M` the is number of destinations, `P` the number of available processors, `ISC` the number of times that the SC model should be executed, and IDC the number of times that the DC model must be computed.

A pre-processor has been written that helps the user set suitable values of N, M, P, ISC and IDC. It reports on array storage required for running the benchmark on hardware implementing 64-bit (8 bytes) floating point precision and also for hardware defaulting to 32-bit (4 bytes) floating point precision. The pre-processor creates a short header file in which the parameters N, M, P, ISC and IDC are specified. Each time a different size of benchmark is to be run or a different number of processors used for a test it is necessary to run the pre-processor code and then compile and link the benchmark code. The pre-processor is written as a serial code thus on some parallel computers needs to be executed on a single processor, if this is not possible it may be necessary to edit and modify a header file sima.h instead of running the pre-processor. It is also possible to run the pre-processor on a sequential computer and then move the sima.h file to a parallel system. The same pre-processor needs to be used before compiling, linking and running a serial benchmark. The problem size used should be selected in such a way so that the benchmark's memory requirements are not bigger than the available memory in order to reduce the impact of paging on virtual memory systems. It follows therefore that not all the standard benchmarks (see Table 1 in Appendix A) can be run on all the hardware.

Compilation of the code should be done with compiler options set at the highest optimisation level available.

Appendix B gives a step by step account of how to run the benchmark and is followed by Appendix C that contains examples of running the benchmark on the serial and parallel architecture.

It should be noted that the benchmark doesn't test the interactive response of a computer system.

9. Verification Test

The benchmark has a built-in results verifier that checks with stored reference values whether or not the numerical results are acceptable. The benchmark is not a test of numerical accuracy. Nevertheless, it verifies the obtained results allowing a 1% relative error for each final sum reduction performed by the code. The program contains a few sum reductions that depending on the order in which they are performed may produce slightly different results when the code runs on a different number of processors. This matter is of no concern here provided the results are within 1% of the *correct* values to verify the benchmark has been properly run.

10. Reporting Results

All SSB benchmark results should contain the information specified below.

The benchmark report should enclose a job's log from the benchmark run. This contains the number of threads used by the benchmark, the size of the dataset used, the number of times the model was computed, the number of processors on which the code was executed, the elapsed time of the computational kernel and the total elapsed time of the execution of the benchmark. Additionally, the report should contain an exact description of the architecture used and its configuration, the maximum number of processors available on the system, the number of processors used for benchmarking, the memory size, the cache size, the memory bandwidth, the operating system and its environment version, the compiler release level, the command used for compilation of the code and the name of the timing routine for the serial code. Moreover, all the modifications of the code that have been done for the optimised run need to be carefully recorded.

Finally, the report should contain the date when the benchmark was carried out and all other necessary information for reproducing results.

Please send the output of the runs to:

j.g.schmidt@leeds.ac.uk

The benchmark results are published periodically as soon as this is possible. All submissions are made available on the World Wide Web.

11. Measuring the Performance of a Parallel System

It is important to realise that a number of factors may affect the performance of a parallel system.

An important indicator of the performance of a parallel system is its scalability. This can be measured with reference to a suitable application. Due to the nature of computations in the SC and DC models, the benchmark code is highly vectorisable and highly parallelisable and it may be used for measuring the scalability of a parallel system.

The benchmark's output reports the elapsed time of the execution of the whole benchmark and the computational kernel. The execution time of the benchmark is a function of the problem size defined by values of N (number of origins), M (number of destinations) and P (number of processors). The performance of a parallel system may be measured by the number of model evaluations done per processor in a fixed amount of time when executing a job over concurrent processors. The size of a job should vary according to the size of the parallel system being benchmarked. Usually, it is necessary for a larger parallel system to be able to run a larger application than a smaller parallel system. Therefore, it is important to be able to determine how the workload changes with the change in a size of the problem being benchmarked. Paragraph 2 gives some indication of the workload being computed by a singly constrained model, however further analysis and experimenting is required to established this accurately.

The DC model implements a considerable number of the communications between processors. This may result in the code not scaling linearly. In both cases as processor speeds increase and problem size is reduced the balance between computation and memory references may well change with loss of scalability effects. This is something to investigate for a particular machine.

The benchmarking results should show the effect of changing the size of the problem and permit the statistical estimation of a wider range of problem sizes without having to run them

Bibliography

Birkin M., Clarke M., George F. (1995), *The Use of Parallel Computers to Solve Nonlinear Spatial Optimisation Problems*, Environment and Planning A

Birkin M., Clarke G., Clarke M, Wilson A., (1996), *Intelligent GIS: Location Decisions and Strategic Planning*, GeoInformation International, Cambridge

Brocklehurst, E.R, (1991). *Survey of Benchmarks*, NPL Report, DITC 192/91

Dongarra J.J.,*Performance of Various Computers Using Standard Linear Equations Software, CS-89-85, Computer Science Department*, University Tennessee, Knoxville, TN 37996-1301, July 1995

Lewis T.G., El Rewini H.,(1992), *Introduction to Parallel Computing*, Prentice Hall, New Jersey

Openshaw, S.(1987), *Some Applications of Supercomputers in Urban and Regional Analysis and Modelling*, Environment and Planning A,19,853-860

Wilson A G (1970), *Urban and Regional Models in Geography and Planning*, Wiley, London

Wilson A G, (1981). *Geography and the Environment*, System Analytical Methods, Wiley, London

Wilson A G and Bennett R J (1985), *Mathematical Methods in Human Geography and Planning*, Wiley, London

Appendix A

Table 1 Suggested Problem Sizes for Social Science Benchmark

Benchmark	Number of Origins (rows) (N)	Number of Destinations (columns) (M)
SSB1	100	100
SSB2	500	500
SSB3	1,000	1,000
SSB4	5,000	5,000
SSB5	10,000	10,000
SSB6	25,000	25,000
SSB7*	50,000	50,000
SSB8*	100,000	100,000
SSB9*	500,000	500,000
SSB10*	2,000,000	2,000,000

* No results verfications are performed for this test.

Appendix B

The suggested procedure for running the serial benchmark on a sequential computer (e.g.PC) is as follows:

1. choose problem size (see Table 1 in Appendix A)
2. compile, link and run the pre-processor i.e. the `simpre.f` program. As a result of running the pre-processor an include file `sima.h` will be created. The include file `sima.h` should contain the following lines:
```
      parameter (n=  1000, m=  1000, nprocs=     1)
      parameter (isc=    100, idc=    10)
```
   where: `n` is the number of origins
   `m` is the number of destinations
   `nprocs` is the number of processors
   `isc` is the number of times that the singly constrained model will be run
   `idc` is the number of times that the doubly constrained model will be run
3. the pre-processor computes approximate array storage for the benchmark. Check that you have enough memory on your system to run the benchmark. If not decrease size of the problem you wish to run.
4. modify the `simtime` function in the `simprog.f` file to contain a procedure a number of seconds from some fixed starting point in the past.
5. compile and link the benchmark (the `simser.f` file for a PC, or `simserial32.f` or `simserial64.f` file for a serial or vector supercomputer)
6. run the benchmark on your system
7. save the output from the benchmark

The suggested procedure for running the parallel benchmark on a parallel system is as follows:

1. choose problem size (see Table in Appendix A)
2. compile, link and run the pre-processor i.e. the `simpre.f` program. As a result of running the pre-processor an include file `sima.h` will be created. The include file `sima.h` should contain the following lines:
```
      parameter (n=  1000, m=  1000, nprocs=   256)
      parameter (isc=    300, idc=    20)
```
   where: `n` is the number of origins
   `m` is the number of destinations
   `nprocs` is the number of processors
   `isc` is the number of times that the singly constrained model will be run
   `idc` is the number of times that the doubly constrained model will be run
3. the pre-processor computes approximate array storage required per processing node. Check that you have enough memory on your system to run the benchmark. If not decrease size of the problem you wish to run.
4. compile and link the benchmark (`simprog32.f` file or `simprog64.f`). The benchmark includes the `mpif.h` include file so you must make sure that the directory to be searched for the MPI include file is specified on the compilation line. Also you need to link the benchmark with the MPI library.
5. on some systems you must assign number of processors on which the job is run. Usually, you need to use the `mpirun` command to run the benchmark on your system.
6. save the output from the benchmark
7. you may choose to run a serial program on one node. There are two ways in which you can do it. You can run the MPI version of the program (`simprog32.f` or `simprog64.f`) on 1 processing node or you can run the supplied serial version of the code (`simserial32.f` or `simserial64.f`) on 1 node of your parallel system. Please note that for a serial run you need to modify the `simtime` function in the `simprog.f` file to contain a procedure a number of seconds from some fixed starting point in the past. It is preferred that you report the results for both runs.

Appendix C

This is an example showing how to run the serial benchmark and output from the run. In this section the following format conventions are used: commands typed by the user are shown in **bold Courier** font; computer output is given in a Courier font.

```
dream1% f77 simpre.f
dream1% a.out
 Please specify the number of trip origins (N)
1000
 Please specify the number of trip destinations (M)
1000
 Please specify the number of processors
 on which the program will be run (NPROCS)
 For the serial run type 1
1
 Please specify how many times you would like
 the singly constrained model to be recomputed (ISC)
1
 Please specify how many times you would like
 the doubly constrained model to be recomputed (IDC)
1
 -----------------------------------------------------

 The program will compute the SIM with       1000 origins and
      1000 destinations.
 The singly constrained model will be recomputed     1 times.
 The doubly constrained model will be recomputed     1 times.
 It will run on     1 processor(s).

 If you would like to change some of the parameters
 please run this program again.

 -----------------------------------------------------

 The parallel program is required to store approx 4899 Kelements
 of arrays on each node.
 It needs         38.3 MBytes  on each node
 to store all elements of arrays using 8 bytes
 for real and integer variables

 and         19.1 MBytes  to store all elements of arrays
 using 4 bytes for real and integer variables.
 -----------------------------------------------------

 The serial version of the program is required to store
 approx       3012 Kelements of  arrays.
 It needs         23.5 MBytes  to store all elements of arrays
 using 8 bytes for real and integer variables
 and              11.8 MBytes  to store all elements of arrays
 using 4 bytes for real and integer variables.
 -----------------------------------------------------------------
dream1% f77 -O3 -64  -mips4  -o simmpi simserial32.f
dream1% simmpi
  SIM  -     Serial Run
 *Origin-constrained model
 *Number of origins (N)=           1000
 *Number of destinations (M)=           1000
 *Trips generated for  1000 origins and   1000 destinations
 *Mean travel length =    74.0168
 *Total trips =    100000.     sum of origin trips =   100000.
 *Number of model evaluations=  1
 *beta= -0.20000E-01 f1= 0.61290280E-01 f2= 0.86539096E-01 c1=62.242287
 *Results verified correctly.

 *Time of data generation=     10.1404791 secs
 *Time of model calculation=      6.1081810 secs
 *Total time =     16.2770996 secs

----------------------------------------

  SIM  -     Serial Run
 *Doubly constrained model
 *Number of origins (N)=           1000
 *Number of destinations (M)=           1000
 *Trips generated for  1000 origins and   1000 destinations
 *Mean travel length =    74.0168
 *Total trips =    100000.     sum of origin trips =    100000.
 *Number of model evaluations=  1
 *beta=-0.20000E-01 f1= 0.10640886 f2= 0.91875328E-01 c1= 0.35251153E-03
 *Results verified correctly.

 *Time of data generation=     10.1651993 secs
 *Time of model calculation=    248.5242310 secs
 *Total time =    258.7185669 secs

----------------------------------------
```

This is an example showing how to run the parallel benchmark and output from the run:

In this section the following format conventions are used: commands typed by the user are shown in **`bold Courier`** font; computer output is given in a `Courier` font.

```
dream1% f77 simpre.f
dream1% a.out
 Please specify the number of trip origins (N)
1000
 Please specify the number of trip destinations (M)
```

```
1000
 Please specify the number of processors
 on which the program will be run (NPROCS)
 For the serial run type 1
2
 Please specify how many times you would like
 the singly constrained model to be recomputed (ISC)
1
 Please specify how many times you would like
 the doubly constrained model to be recomputed (IDC)
1
 -----------------------------------------------------

 The program will compute the SIM with        1000 origins and
      1000 destinations.
 The singly constrained model will be recomputed     1 times.
 The doubly constrained model will be recomputed     1 times.
 It will run on     2 processor(s).

 If you would like to change some of the parameters
 please run this program again.

 -----------------------------------------------------

 The parallel program is required to store approx 2212 Kelements
 of arrays on each node.
 It needs              17.3 MBytes  on each node
 to store all elements of arrays using 8 bytes
 for real and integer variables

 and               8.6 MBytes  to store all elements of arrays
 using 4 bytes for real and integer variables.
 -----------------------------------------------------

 The serial version of the program is required to store
 approx       3012 Kelements of  arrays.
 It needs          23.5 MBytes  to store all elements of arrays
 using 8 bytes for real and integer variables

 and              11.8 MBytes  to store all elements of arrays
 using 4 bytes for real and integer variables.
 -----------------------------------------------------
dream1% f77 -O3 -64 -mips4 -I/home/dream/mpi/mpich/include -o simmpi
      -L/home/dream/mpi/mpich/lib/IRIX64/ch_shmem -lmpi simprog32.f
dream1% mpirun -np 2 simmpi
*SIM_mpi_fortran77 version
   2   1   3
 *Origin-constrained model
 *Number of origins (N)=           1000
 *Number of destinations (M)=           1000
 *Number of threads =    2
 *Mean travel length =    74.0168
```

```
*Total trips =    100000.      sum of origin trips =    100000.
*Number of model evaluations=  1
*beta= -0.20000E-01 f1= 0.61290280E-01 f2= 0.86539096E-01 c1=62.242287
*Results verified correctly.

*Time of data generation=      5.6258709 secs
*Time of model calculation=     3.1752800 secs
*Total time =       8.8212759 secs

----------------------------------------

*SIM_mpi_fortran77 version
  4   1   3
*Doubly-constrained model
*Number of origins (N)=          1000
*Number of destinations (M)=          1000
*Number of threads =   2
*Mean travel length =   74.0168
*Total trips =    100000.      sum of origin trips =    100000.
*Number of model evaluations=  1
*beta= -0.20000E-01 f1= 0.10640886 f2= 0.91875328E-01 c1= 0.35251153E-03
*Results verified correctly.

*Time of data generation=      5.4893681 secs
*Time of model calculation=    145.1070700 secs
*Total time =     150.6217051 secs
```

A Social Science Benchmark (SSB/1) Code for Serial, Vector, and Parallel Supercomputers

Stan Openshaw, Centre for Computational Geography, School of Geography, University of Leeds, Leeds LS2 9JT
email: stan@geography.leeds.ac.uk

Joanna Schmidt, University Computing Service, University of Leeds, Leeds LS2 9JT
email: j.g.schmidt@leeds.ac.uk