LABORATORY FOR
CONCURRENT COMPUTING SYSTEMS

COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne Institute of Technology
John Street, Hawthorn 3122, Victoria, Australia.

Some Parallelisation Issues in a Triangular Matrix Problem

Technical Report 31-022

P.S. Chang
pau@stan.xx.swin.oz(.au)

G.K. Egan
gke@stan.xx.swin.oz(.au)

Version 1.0 (November 16, 1990)
Version 2.0 (December 16, 1990)
Version 3.0 (December 21, 1990)

Abstract

This report describes an application study in explicit and implicit parallel computing for a simplified electrical transmission line which is modelled in a triangular structural form to evaluate the potential distribution in its cross section. A traditional (sequential) formulation and a parallel formulation of the numerical algorithm to Laplace Equation, which solves this problem, are described. They are implemented serially, and the parallel formulation is also implemented in parallel. The codes are executed on an IBM RS6000/530 workstation, and a 4 XPC processor based, as well as a 20 APC processor based, Encore Multimax multiprocessors. The results of the complete runs show that the formulation for parallel implementation is competitive with the sequential traditional formulation. The benchmarks on the two Multimax environments show scalable, and close to ideal, speedup performance for the explicit implementation in C, with low parallelisation overhead. The speedup performance of the SISAL implementation is poor in the computations for triangular matrices because the loop slicing scheme implemented in OSC fails to distribute the decomposed loops more equally. This scheme, and a solution to this load balancing problem by way of loop splitting, are critically analysed and their results discussed. The analysis shows that loop splitting is a better general solution than loop slicing to gain good code performance and parallel processing support in shared memory multiprocessor environments similar to that of Encore Multimax.

Some Parallelisation Issues in a Triangular Matrix Problem

P.S. Chang
G.K. Egan

Version 1.0 (November 16, 1990)
Version 2.0 (December 16, 1990)
Version 3.0 (December 22, 1990)

Introduction

This project was originated from an attempt to utilise the syncronization primitives in the parallel programming library (UMAX 4.3 and UMAX V) of Encore Multimax to perform explicit parallel computing on a share memory MIMD multiprocessor.

The engineering problem being investigated is a numerical algorithm for Laplace Equation (it was part of the author's second year undergraduate programming project (1985) in Electrical Engineering at the University of Melbourne). In this case, the equation is used to evaluate the potential distribution in an electrical transmission line. The simplified transmission line is assumed to have a rectangular cross section. And only one quarter of the section is focused on due to symmetry, resulting in a model with a triangular structure.

Two implementations of the numerical algorithm are being examined. The first is a traditional method targeted for conventional sequential computers. This version is potentially sequential, and hence is unsuitable for parallelisation. As a result, another version was formulated to expose the inherent large amounts of parallelism, which can be effectively exploited in a multiprocessor environment.

The numerical algorithm for Laplace Equation is implemented in C, and explicitly parallelised also in C. The implicit implementation in SISAL [1] is a straightforward translation from the C parallel version, with minor modifications. The results of these implementations, which are obtained from an IBM RS6000/530 and two Encore Multimax multiprocessors, are critically analysed and compared. The degradation in the speedup performance of SISAL and OSC [2] is observed and examined. The dominant factor to the degradation due to loop scheduling, and a solution to this problem are presented.

1. Laplace Equation

Laplace Equation is Poisson's Equation equated to zero.

Poisson's Equation: $\nabla^2 V = f(V)$
Laplace Equation: $\nabla^2 V = 0$

This equation can be used to analyse and solve many engineering problems, such as modeling streamline in fluid flow and electrical field or potential distribution in a transmission line.

1.1 Formulation of Numerical Algorithm

A numerical solution to this equation is via an iterative method, in which calculations are repeated until the values of the grid points converge. Traditionally, this is formulated as:

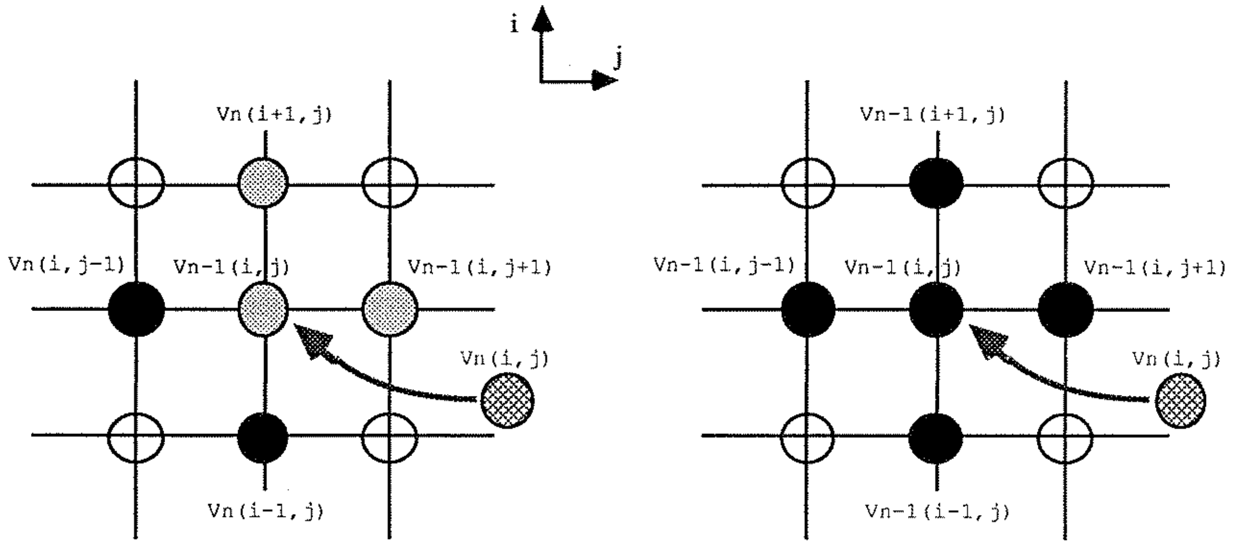
$$V_n(i,j) = [\, V_n(i+1,j) + V_n(i,j+1) + V_{n-1}(i-1,j) + V_{n-1}(i,j-1) - 4.0\, V_{n-1}(i,j) \,] \;*\; \frac{RF}{4.0} \;+ V_{n-1}(i,j)$$

where RF is a relaxation factor arbitrarily chosen to help speeding up the convergence of the results, and the relative North, East, South, West and Centre points of the n-th iteration can be visualised as *(i,j)*-cordinates as shown in Figure 1a. The computation takes turn from the bottom to the upper rows (*i*) and propagates from the left to the right columns (*j*) in every row. Hence, the values of the West and the South points are in fact of the current (n-th) iteration.

If the strategy of parallelisation is to compute from the left to the right columns sequentially for all rows simultaneously (in other words, to parallelise at the row level), then this formulation will cause *write-read* race conditions which constrain parallelisation. It is thus obvious now that another formulation which is suitable for parallelisation is needed. One such formulation is

$$V_n(i,j) = [\, V_{n-1}(i+1,j) + V_{n-1}(i,j+1) + V_{n-1}(i-1,j) + V_{n-1}(i,j-1) - 4.0\, V_{n-1}(i,j) \,] \;*\; \frac{RF}{4.0} \;+ V_{n-1}(i,j)$$

as described in Figure 1b. Using two separate storages for V_n and V_{n-1}, this method is free of any *write-read* race condition, and therefore it is suitable for parallelisation (with the tradeoff that the storage requirement is doubled).



(a) Sequential Formulation *(b) Parallel Formulation*

Figure 1: Formulation of the iterative method.

1.2 An Engineering Application

The chosen engineering application of Laplace Equation, in this case, is to evaluate the potential distribution in a simplified electrical transmission line. The cross section of the transmission line is assumed, for the sake of simplicity, to be rectangular as shown in Figure 2a. Owing to symmetry, only one quarter of the section needs to be investigated; the problem to be delt with is thus a triangular structure (Figure 2b) which may be represented by triangular matrices.

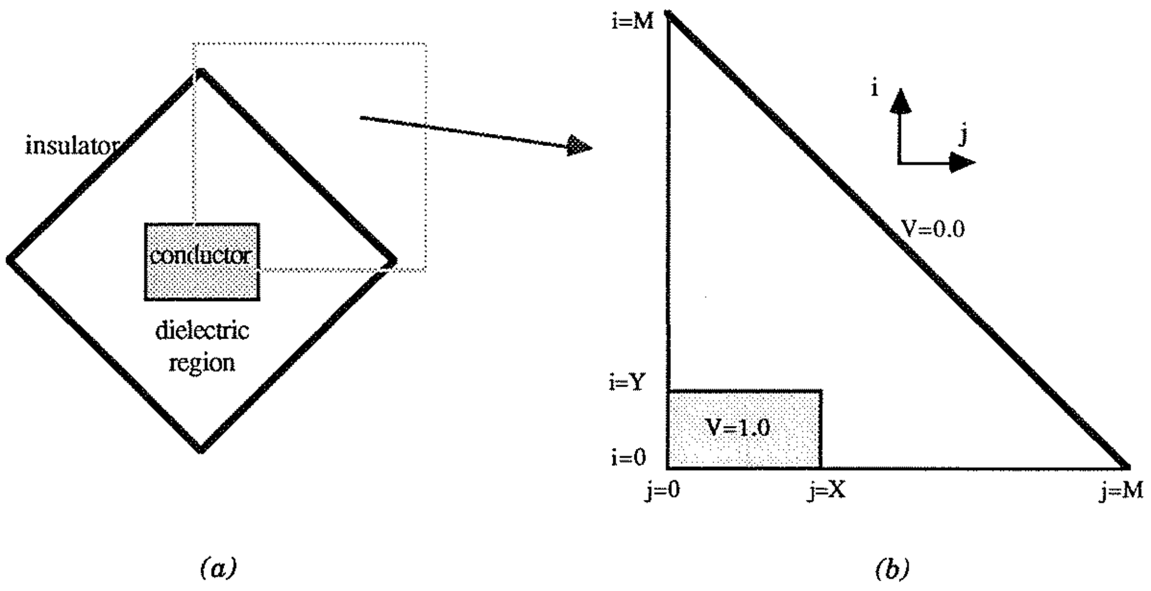


Figure 2: Modeling transmission line

In this model, the conductor and insulator regions have constant potentials which are normalised to `1.0` and `0.0` respectively. This leaves the dielectric region to be computed. Referring to Figure 2b, there are three conditions (areas) in the evaluation i.e. the vertical border on the extreme left (`j=0 & i>Y`) whose East points are the mirror of their West points , the horizontal border at the bottom (`i=0 & j>X`) whose North points are the mirror of their South points, and the rest of the dielectric region which follows the diagram in Figure 1. The starting values of the points in this region are initialised to `0.5`. The structure of the transmission line is set to be `M=1000`, `X=6` and `Y=10` (these values may be altered). And finally, the relaxation factor `RF` is set to `0.9`.

The terminating condition of the computation is the difference between the newly evaluated values of <u>all</u> grid points and their corresponding old values, which must be less than `0.0001` (this value is variable, depending on the required accuracy of the final results).

2. Implementation of Formulations

The sequential and parallel formulations of the numerical algorithm for Laplace Equation are implemented in the C programming language. The parallel formulation has also been implemented in SISAL (Appendix).

2.1 Sequential Formulation

In `tradmat.c`, the traditional (sequential) formulation is implemented using only one matrix, ie V[i][j], to express the solution as illustrated in Figure 1a. Another equivalence is `tradpoint.c`, in which pointers to arrays, i.e. `(*rowV)[j]`, are used in place of the matrix so as to speedup the execution of this sequential formulation.

2.2 Parallel Formulation

The parallel formulation has been implemented in `seqmat.c`, `seqpoint.c`, `parmat.c` and `parpoint.c`.

2.2.1 Sequential Implementation

`seqmat.c` is a sequential implementation of the parallel formulation illustrated in Figure 1b, in that none of its program parts involves creation and syncronisation of multiple processes, and job distribution for these processes. In this code, the new and old values of the potentials are stored in `newV[i][j]` and `V[i][j]` matrices respectively. (`(*rownew)[j]` is used in place of `newV[i][j]` in `seqpoint.c`, the corresponding pointers-to-arrays version). By swapping the first pointers of `(*rowV)[j]` and `(*rownew)[j]` before the computations are repeated in the next iterations, the old and new matrices have their values swapped in negligible time (the parallel formulation is thus costly in standard FORTRAN implementation because the former does not provide such luxury).

2.2.2 Explicit Parallel Implementation in C

The parallel implementation of the parallel formulation, `parmat.c` and `parpoint.c`, are coded using the synchronization primitives of the parallel programming library (`parallel.h`) on Encore's UMAX 4.3 and UMAX V operating systems. These primitives are used in creating shared memory spaces, in `createshare()`, for the matrices and other shared variables. Child processes (there are `nproc` number of them) are created in `spawn()` and merged in `join()`. Once the program has called `spawn()`, the program section down to the call to `join()`, and the non-shared data, are replicated for each created child process. By such, the workload is decomposed to all processes. However, work is shared only at the loop level using Loop Splitting (Figure 8):

```
for (i=id; i<=M; i+=nproc)
```

2.2.3 Implicit Parallel Implementation in SISAL

Except for some minor modifications to checking for program termination condition, the (parallel) implementation in SISAL, which represents a matrix as an array of arrays, is relatively much simpler than that in C. It is almost a straightforward translation from the sequential implementation in `seqmat.c`; in `sisal.sis`, `OLD V[i][j]` is used to naturally represent the old values of `V[i][j]`. Furthermore, in SISAL, one does not have to explicitly deal with shared memory allocation, process creation and termination, loop decomposition, work scheduling and process synchronization as they are otherwise necessary in the explicit parallel implementation in C because these are all conducted by SISAL's compiler, OSC, and OSC's dynamic routines.

3. Performance

These codes have been executed on a 4 XPC processor based (4 MIPS per processor) and a 20 APC processor based (2 MIPS per processor) Encore Multimax multiprocessor, as well as an IBM RS6000/530 (30 MIPS) workstation. The Multimax multiprocessors are useful in the research on the prospective performance of (shared memory) parallel computations although their processors are not high performance RISCs like the RS6000 processors. The latter is therefore used here as a contrast to indicate the realistically achievable run time and speedup performance of multiprocessor systems which are based on high performance processors of the class, or better, of the RS6000.

3.1 Performance Comparisons of Traditional and Parallel Formulations

As already described, the computations are iterated until the results converge to less than a threshold of `0.0001` with RF set to `0.9` (this RF value has been checked to ensure good convergence speed for both formulations). Thus it is important to compare the convergence speed of both formulations in terms of the number of iterations (`iter`) required for convergence, and their computation times for *complete runs* (programs execute until the results converge) of their programs. For these comparisons, the program termination condition in the codes is:

```
while (repeat>0)
iterate;
```

where `repeat` is set to 0 if the values of all points satisfy the termination threshold, and to 1 otherwise.

The results obtained from the IBM RS6000/530 and the 4 XPC processor based Encore Multimax are tabulated in Table 1. They indicate that the parallel formulation is competitive, in every sense, with the traditional formulation (except that the storage requirement is twice, of course). The former actually converges approximately 2% faster. However, although it completes in also 2% less time on the RS6000 machine, it surprisingly takes 2% more time on the 4 XPC based Multimax.

On the other hand, the SISAL implementation of the parallel formulation runs significantly slower on the RS6000 machine because, currently, the IF1 code optimisation routines has not yet been successfully installed in the OSC compiler on the machine. The result would have been (more) competitive with the C implementations otherwise, as indicated by the SISAL run time on the 4 XPC Multimax which is over 2 hours faster than `parpoint.c`.

machines	RS6000/530				4 XPC processor Encore			
formulations	traditional		parallel		traditional	parallel		
codes	tradmat.c	tradpoint.c	seqpoint.c	sisal.c	tradpoint.c	seqpoint.c	parpoint.c	sisal.sis
iter	1835	1835	1801	1802	1835	1801	1802	1802
time(s)	3343	3341	3276	5501	36784	37568	38894	30607

Table 1: Results of complete runs on two different machines

3.2 Performance on Encore Multimax Multiprocessors

The performance evaluation of the complete runs on the Multimax multiprocessors serves to demonstrate the time significance of parallelism in codes and parallelisation of these codes. On the other hand, the evaluation of the one-iteration runs is meant for detailed investigation of the parallel processing performance of these codes.

3.2.1 Complete Runs

A complete run of each of the codes developed requires a long execution time on the two Multimax multiprocessors. `parpoint.c`, for example, takes nearly 11 hours on one processor of the 4 XPC Multimax, and is expected to take twice this time on the 20 APC Multimax. As a result, the benchmark of the complete runs has been performed only on the faster Multimax (being the 4 XPC Multimax), and using only the code that runs faster (being `seqpoint.c`, `parpoint.c` and `sisal.sis`). The results of the complete runs on Table 2 amply indicate the impact of parallel processing support on reducing computation time. With 94% utilisation of 4 processors (Figure 4), the run time of `parpoint.c` (Figure 3) has been significantly reduced down to approximately 3 hours (264% faster than its single CPU run time).

	seqpoint.c	parpoint.c			sisal.sis	
# Proc	time(s)	time(s)	S	Sco	times(s)	S
1	36784 (10.2 hours)	38894 (10.8 h)	1.00	0.95	30607 (8.5h)	1.00
2	-	20310 (5.6 h)	1.92	1.81	22498 (6.2h)	1.36
3	-	13900 (3.9 h)	2.80	2.65	17528 (4.9h)	1.75
4	-	10403 (2.9 h)	3.74	3.54	14903 ((4.1 h)	2.05

Table 2: Results of complete runs on a 4 XPC processor based Encore Multimax multiprocessor

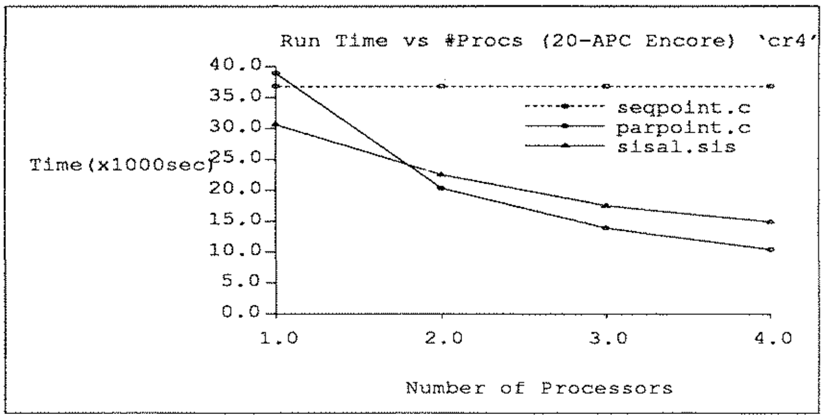


Figure 3: Execution curves for complete runs on the 4 XPC based Multimax

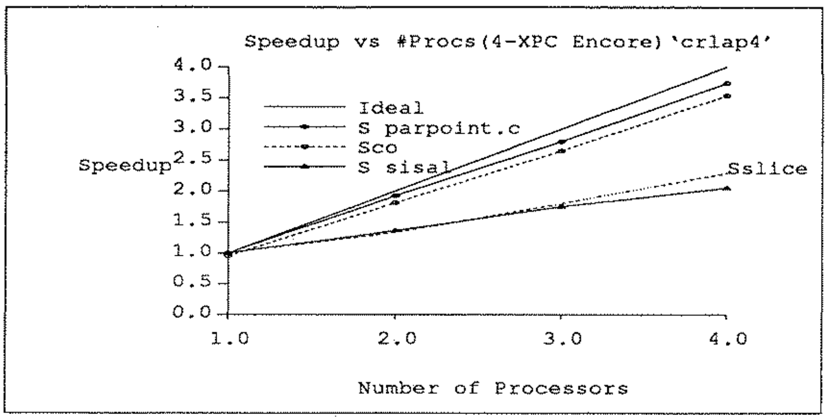


Figure 4: Speedup curves for complete runs on the 4 XPC based Multimax

3.2.1 One-Iteration Runs

	tradmat.c	tradpoint.c	seqmat.c	parmat.c			seqpoint.c	parpoint.c			sisal.sis (SISAL 1.2)		Speedup estimates	
# Proc	Time(s)	Time(s)	Time(s)	Time(s)	S	Sco	Time(s)	Time(s)	S	Sco	Time(s)	SU	Sslice	Ssplit
1	81.0	55.0	86.6	88.8	1.00	0.98	58.8	59.8	1.00	0.98	52.6	1.00	1.00	1.00
2	-	-	-	44.7	1.99	1.94	-	30.2	1.98	1.95	41.04	1.28	1.33	2.00
3	-	-	-	30.0	2.96	2.89	-	20.6	2.90	2.85	32.22	1.63	1.80	2.99
4	-	-	-	22.7	3.91	3.81	-	15.5	3.86	3.79	26.70	1.97	2.29	3.99
5	-	-	-	18.2	4.88	4.76	-	12.7	4.71	4.63	23.58	2.23	2.78	4.98
6	-	-	-	15.6	5.69	5.55	-	10.6	5.64	5.55	20.80	2.53	3.27	5.97
7	-	-	-	13.6	6.53	6.37	-	9.20	6.50	6.39	18.78	2.80	3.77	6.96
8	-	-	-	11.8	7.53	7.34	-	8.10	7.38	7.26	17.52	3.00	4.27	7.94
9	-	-	-	10.5	8.46	8.25	-	7.20	8.31	8.17	16.52	3.18	4.77	8.93
10	-	-	-	9.50	9.35	9.12	-	6.90	8.67	8.52	15.52	3.39	5.26	9.91
11	-	-	-	9.00	9.87	9.62	-	5.90	10.10	9.97	14.72	3.57	5.76	10.89
12	-	-	-	8.10	10.96	10.69	-	5.80	10.31	10.14	14.12	3.72	6.26	11.87
13	-	-	-	7.60	11.68	11.39	-	5.40	11.07	10.89	13.52	3.89	6.76	12.85
14	-	-	-	7.30	12.16	11.86	-	4.90	12.20	12.00	13.06	4.03	7.26	13.82
15	-	-	-	7.00	12.69	12.37	-	4.60	13.00	12.78	12.82	4.10	7.76	14.79
16	-	-	-	6.30	14.10	13.75	-	4.40	13.59	13.36	12.66	4.15	8.26	15.76

Table 3: Results gathered from the 20 APC processor based Encore Multimax

	tradmat.c	tradpoint.c	seqmat.c	parmat.c			seqpoint.c	parpoint.c			sisal.sis SISAL 1.2 V2.0	
# Proc	Time(s)	Time(s)	Time(s)	Time(s)	S	Sco	Time(s)	Time(s)	S	Sco	Time(s)	S
1	31.6	22.6	34.6	37.5	1.00	0.92	26.6	27.4	1.00	0.97	15.35	1.00
2	-	-	-	19.9	1.88	1.74	-	14.7	1.86	1.81	11.39	1.35
3	-	-	-	13.9	2.70	2.49	-	10.1	2.71	2.63	8.59	1.79
4	-	-	-	10.3	3.64	3.36	-	8.10	3.38	3.28	8.02	1.91

Table 4: Results gathered from the 4 XPC processor based Encore Multimax

All the implemented codes spend most of their life time iterating in the loop iteration section. For the purpose of investigation into the parallel processing support given to the parallel codes with up to 16 processors, it suffices therefore to benchmark these codes with only one loop iteration. This eases the benchmark on the 20 APC based Multimax. Thus, the program termination condition of these codes for this benchmark is:

```
while (iter<=0)
iterate;
```

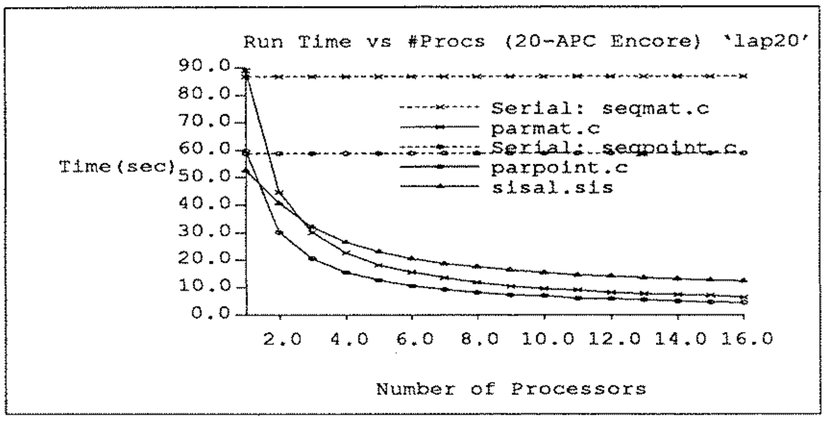


Figure 5: Execution time curves in the 20 APC processor based environment

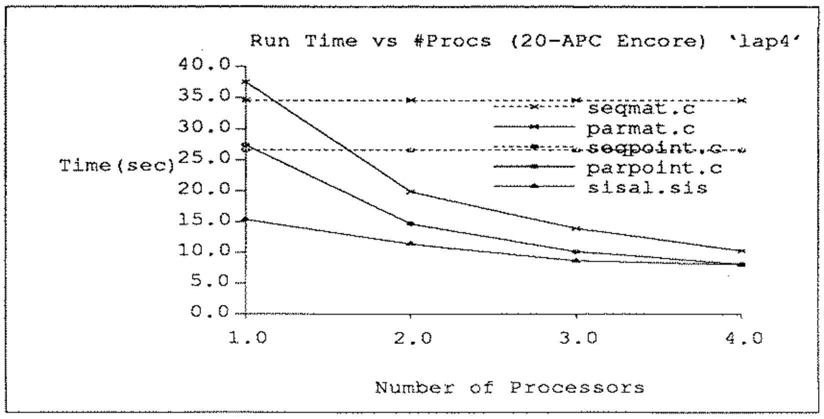


Figure 6: Execution time curves in the 4 XPC processor based environment

The execution results of these codes on two separate Encore Multimax multiprocessor environments are tabulated in Table 3 and Table 4. The run times and speedups for the two environments as a function the number of processors are plotted in Figures 5 to 8. The versions of OSC compilers on the 20 APC and 4 XPC processor based Multimax environments are "SISAL 1.2" (perhaps V1.0 ??) and "SISAL 1.2 V2.0" respectively.

It is worth noted also that the 20 APC based Multimax has abundantly free processors to service other jobs in the system at the time of the benchmark while the 4 XPC based Multimax has not. Hence, the results gathered from the latter may be less consistent or accurate than they should be.

3.2.2.1 Execution Time

As a result, the comparisons in the single processor run times of `seqmat.c` with `parmat.c`, and that of `seqpoint.c` with `parpoint.c` in Table 3 (20 APC based) show that the overhead due to parallelisation is gratifyingly low, being around 2.5% and 2% respectively. And the same comparisons in Table 4 (4 XPC based) show the respective parallelisation overheads of 8% and 3%.

The update in-place optimisation and the adoption of microtasking in OSC has improved the run time of SISAL codes. This is demonstrated by the execution time curve of `sisal.sis` in Figure 5 which shows that, on the single processor run, the implicit SISAL implementation runs 12% faster than the more efficient version of the explicit C implementation (`parpoint.c`). The new version of OSC performs even faster, being 44% for similar comparison, as shown in Figure 6.

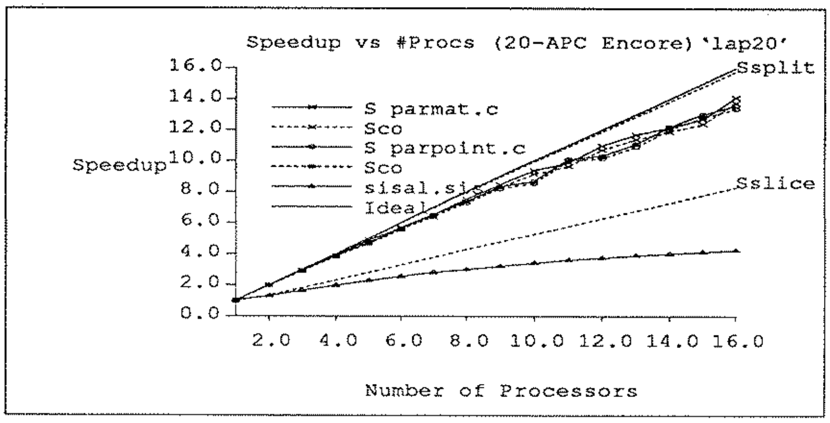


Figure 7: Speedup curves in the 20 APC processor based environment

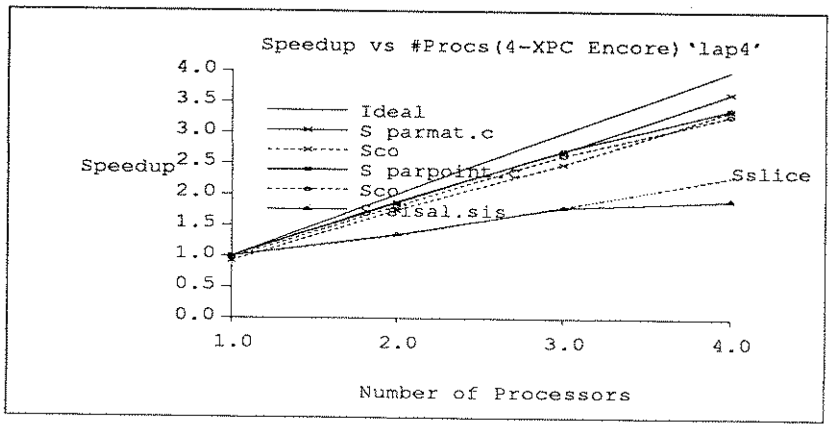


Figure 8: Speedup curves in the 4 XPC processor based environment

3.2.2.2 Speedup

The implementations in C (`parmat.c` and `parpoint.c`) are effectively parallelised with the speedup curves in Figures 7 and 8 showing very good parallel processing support provided by the two Multimax multiprocessors. Owing to the effect of different system configuration described earlier, the speedup performance on the 4 XPC based Multimax could be expected to be as good as that on the other Multimax if it has the same *ideal* configuration.

Nevertheless, the results have clearly indicated that more processors may be incorporated to obtain further scalable speedups with very high machine utilisation, thus the success of the explicit parallelisation in C.

While the single processor run time of the SISAL code is significantly faster than the C equivalence, it is unfortunate that SISAL's speedup performance is shown to be poor in this case, and worse when more processors are incorporated (only approximately 30% of the C codes' speedups at 16 processor, in Figure 7). The vital factor contributing to the poor performance of the SISAL code is the inefficiency of the job scheduling scheme, called Loop Slicing, adopted by OSC to decompose and distribute, in this case, uneven loops which deal with triangular matrices. This leads to the analysis in the next section.

The results also indicate the characteristics of shared memory parallel processing as discussed in [3] such as the scalable improvement with multiple processors, Amdahl's effect, and the effect of memory contention in a shared memory environment when a large number of processors are incorporated.

4.0 Load Balancing and Loop Scheduling in Triangular Matrix Problem

In a triangular matrix problem, Loop Slicing scheme generates imbalance loads for all processes. Assuming that no other overheads are present, Figure 9 illustrates that the speedup achievable with multiple processors is solely dependent on the process which acquires the heaviest load, resulting in a much degraded estimated speedup S_{slice} of

$$S_{slice} = \frac{(nproc)^2}{2(nproc-2)+3}$$

where *nproc* refers to the number of processors. This estimation is supported by the curves of `sisal.sis` and S_{slice} shown in Figures 5 and 6.

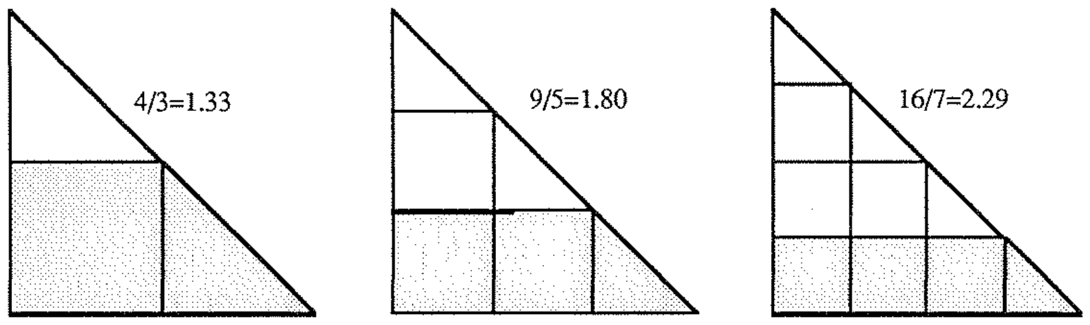


Figure 9: Estimation of speedup S_{slice} from Loop Slicing for a triangular matrix problem.

A good solution recommended to this problem, which is vital to parallel programming in SISAL, is by implementing Loop Splitting in the job scheduling scheme in OSC. Using this scheme, the job of a loop such as

```
for (i=0; i<M; i++)
{ for (j=0; j<M-i; j++)
   ....etc....
}
```

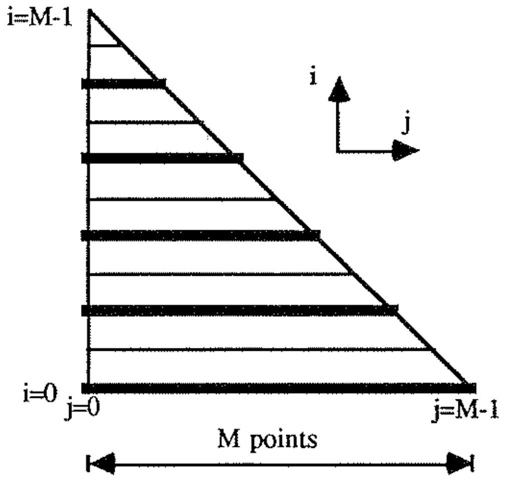


Figure10: Estimation of speedup S_{split} from Loop Splitting for a triangular matrix problem.

can be distributed more evenly to all processes, as shown in Figure 10, if it is splitted along `i` as

```
for (i=id; i<M; i+=nproc)
{ for (j=0; j<M-i; j++)
   ....etc....
}
```

where `id` the processes' i.d. numbers ranging from 0 to `nproc`-1. In this case, the total number of points dealt with is

$$\sum_{p=1}^{M} p \qquad \text{or} \qquad \frac{M(M+1)}{2}$$

And the longest chain (the most number of points given to a particular process) is

$$\sum_{p=M}^{1} p \qquad ; \qquad p\text{-=}nproc$$

which is equivalent, in C, to

```
sum=0;
for (p=M; p>=1; p-=nproc)
sum+=p;
```

The estimated speedup S_{split}, assuming no parallelisation overhead, is then

$$S_{split} = \frac{\frac{M(M+1)}{2}}{\sum_{p=M}^{1} p \qquad ; \; p\text{-=}nproc}$$

which almost coincides with the ideal speedup for nproc ranging from 1 to 16. As a result, as has been shown in earlier sections and in Figures 4, 8 and 10, the explicit C implementations which adopted Loop Splitting exhibit much better speedup performance than the present SISAL implementation.

Conclusions

In this application study in explicit and implicit parallel computing, a simplified electrical transmission line model has been analysed, in a triangular structure, to evaluate the potential distribution in the model. A numerical algorithm to Laplace Equation, which solves this problem, has been investigated. A traditional (sequential) formulation and a parallel formulation of the algorithm have been described. These formulations have been implemented serially, and the parallel formulation has also been implemented in parallel. The codes have been executed on an IBM RS6000/530 workstation, and a 4 XPC processor based, as well as a 20 APC processor based, Encore Multimax multiprocessor. The results of the complete runs show that the formulation for parallel implementation is competitive with the sequential traditional formulation. The benchmarks on the two Multimax environments show scalable, and close to ideal, speedup performance for the explicit implementation in C, with low parallelisation overhead, thus indicating that the explicit parallelisation attempt has been successful. The speedup performance of the SISAL implementation of the parallel formulation is, however, poor in this application (which is modelled by triangular matrices) because the loop slicing scheme implemented in OSC fails to distribute the decomposed loops more equally. This scheme, and a solution to this load balancing problem by way of loop splitting, have been critically analysed and their results discussed. The analysis has concluded that loop splitting is a better general solution than loop slicing to gain good code performance and parallel processing support in shared memory multiprocessor environments similar to that of Encore Multimax.

Acknowledgements

I would like to thank the Department of Communication and Electrical Engineering at RMIT for kindly providing access to its 20 APC processor based Encore Multimax, and Professor Greg Egan of LCCS for his contribution to discussions on the issue of loop slicing and the general aspects of parallel processing.

References

[1] McGraw J. et al., "SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2", Memo 146, Lawrence Livermore National Laboratory, March 1985.

[2] Cann D., "Compilation Techniques for High Performance Applicative Computation", Technical Report CS-89-108, Colorado State University, May 1989.

[3] P.S. Chang, "Implementation of a Numerical Weather Prediction Model in SISAL", *Master's Thesis*, RMIT Victoria University of Technology, 1990; also Technical Report 31-017, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, June 1990.

Appedix (i) tradmat.c

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

#define M 1000
#define X 6
#define Y 10
#define RFAC 0.9

typedef float GRID[M+6][M+6];
typedef float (*RP)[M+6];

float   LAPLACE(North,West,South,East,Centre)
float    North,West,South,East,Centre;
{        return (North+West+South+East - 4.0*Centre) * RFAC/4.0 + Centre;
}

int compute(V, incomplete)
GRID V;
int    incomplete;
{ int    i, j;
  float oldV, LAPLACE();

for (i=0; i<M; i++)
{            for (j=0; i+j<M; j++)
             { oldV=V[i][j];
              if (j==0 & i>Y)   V[i][0]=LAPLACE( V[i+1][0], V[i][1], V[i-1][0], V[i][1], V[i][0] );
              else if (i==0 & j>X)          V[i][j]=LAPLACE( V[i+1][j], V[i][j-1], V[i+1][j], V[i][j+1], V[i][j] );
              else if (i>Y | j>X)           V[i][j]=LAPLACE( V[i+1][j], V[i][j-1], V[i-1][j], V[i][j+1], V[i][j] );
               if ( fabs(V[i][j]-oldV) >= 0.0001 ) incomplete=1;
             }
}
return(incomplete);
}

void initialise(V)
GRID V;
{  int     i, j;
   float   Vcore=1.0, Vout=0.0, Vgap=(Vcore+Vout)/2.0;
for (i=0; i<=M; i++)
             for (j=0; i+j<=M; j++)
             {  if (i+j>=M) V[i][j]=Vout;
             else if (i<=Y & j<=X) V[i][j]=Vcore;
             else V[i][j]=Vgap;
             }
}

main()
{ GRID V;
  RP    rowV;
   int  compute(), repeat, i, j, iter;

rowV=V;
initialise(rowV);
iter=1;
repeat=1;

/* use 'repeat' for a complete run */
/*while (repeat>0)*/
while(iter<=1)
  {          iter++;
             repeat=0;
             repeat=compute(rowV,repeat);
  }
printf("iter=%d\n",iter-1);
}
```

Appendix (ii) tradpoint.c

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

#define M 1000
#define X 6
#define Y 10
#define RFAC 0.9

typedef float GRID[M+6][M+6];
typedef float (*RP)[M+6];

void initialise(rowV)
RP  rowV;
{  int    i, j;
   float Vcore=1.0, Vout=0.0, Vgap=(Vcore+Vout)/2.0;
for (i=0; i<=M; i++)
 {         for (j=0; i+j<=M; j++)
           if (i+j>=M)                (*rowV)[j]=Vout;
           else if (i<=Y & j<=X) (*rowV)[j]=Vcore;
           else                        (*rowV)[j]=Vgap;
 rowV++;
 }
}

float LAPLACE(North,West,South,East,Centre)float North,West,South,East,Centre;
{    return (North+West+South+East - 4.0*Centre) * RFAC/4.0 + Centre;
}

int compute(rowV,incomplete)
RP rowV;
int incomplete;
{ int    i,j;
 float oldV, LAPLACE();
for (i=0; i<M; i++)
   {  for (j=0; i+j<M; j++)
     {      oldV=(*rowV)[j];
           if (j==0 & i>Y)   (*rowV)[0]=LAPLACE( (*(rowV+1))[0], (*rowV)[1],
                                                        (*(rowV-1))[0], (*rowV)[1], (*rowV)[0] );
           else if (i==0 & j>X)   (*rowV)[j]=LAPLACE( (*(rowV+1))[j], (*rowV)[j-1],
                                                        (*(rowV+1))[j], (*rowV)[j+1], (*rowV)[j] );
           else if (i>Y | j>X)   (*rowV)[j]=LAPLACE( (*(rowV+1))[j], (*rowV)[j-1],
                                                        (*(rowV-1))[j], (*rowV)[j+1], (*rowV)[j] );
           if ( fabs((*rowV)[j]-oldV) >= 0.0001 ) incomplete=1;
     }
   rowV++;
   }
return(incomplete);
}

main()
{  int       i, j, repeat, iter, compute();
  GRID    V;
  RP     rowV;
rowV=V;
initialise(rowV);
repeat=1;
iter=1;
/* use 'repeat' for a complete run */
/*while(repeat>0)*/
while(iter<=1)
    {      repeat=0;
           iter++;
           repeat=compute(rowV,repeat);          }
printf("iter = %d\n",iter-1);
}
```

Appendix (iii) seqmat.c

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

#define M 1000
#define X 6
#define Y 10
#define RFAC 0.9

typedef float GRID[M+6][M+6];
typedef float (*RP)[M+6];

float LAPLACE(North,West,South,East,Centre)
float North,West,South,East,Centre;
{    return (North+West+South+East - 4.0*Centre) * RFAC/4.0 + Centre;
}

int    compute(V, newV, incomplete)
GRID V, newV;
int    incomplete;
{ int    i, j;
 float LAPLACE();
for (i=0; i<M; i++)
  for (j=0; i+j<M; j++)
  {
  if (j==0 & i>Y)    newV[i][0]=LAPLACE( V[i+1][0], V[i][1], V[i-1][0], V[i][1], V[i][0] );
  else if (i==0 & j>X)           newV[i][j]=LAPLACE( V[i+1][j], V[i][j-1], V[i+1][j], V[i][j+1], V[i][j] );
  else if (i>Y | j>X)            newV[i][j]=LAPLACE( V[i+1][j], V[i][j-1], V[i-1][j], V[i][j+1], V[i][j] );
   if ( fabs(newV[i][j]-V[i][j]) >= 0.0001 ) incomplete=1;
  }
return(incomplete);
}

void initialise(V, newV)
GRID V, newV;
{  int    i, j;
   float Vcore=1.0, Vout=0.0, Vgap=(Vcore+Vout)/2.0;
   for (i=0; i<=M; i++)
            for (j=0; i+j<=M; j++)
            {  if (i+j>=M)         { V[i][j]=Vout;     newV[i][j]=Vout; }
              else if (i<=Y & j<=X)  { V[i][j]=Vcore;          newV[i][j]=Vcore; }
              else                 { V[i][j]=Vgap;     newV[i][j]=Vgap; }
            }
}

main()
{  GRID V, newV;
   RP    rowV, rownew, tem;
   int   compute(), repeat, i, j, iter;

rowV=V;
rownew=newV;
initialise(rowV, rownew);
iter=1;
repeat=1;
/* use 'repeat' for a complete run */
/*while (repeat>0)*/
while(iter<=1)
   {        iter++;
            repeat=0;
            tem=rownew;
            rownew=rowV;
            rowV=tem;
            repeat=compute(rowV,rownew,repeat);          }
printf("iter=%d\n",iter);
}
```

Appendix (iv) seqpoint.c

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

#define M 1000
#define X 6
#define Y 10
#define RFAC 0.9

typedef float GRID[M+6][M+6];
typedef float (*RP)[M+6];

void initialise(rowV, rownew)
RP   rowV, rownew;
{ int    i, j;
  float Vcore=1.0, Vout=0.0, Vgap=(Vcore+Vout)/2.0;

for (i=0; i<=M; i++)
     {      for (j=0; i+j<=M; j++)
            {  if (i+j>=M)              { (*rowV)[j]=Vout;          (*rownew)[j]=Vout; }
            else if (i<=Y & j<=X)       { (*rowV)[j]=Vcore;         (*rownew)[j]=Vcore; }
            else                        { (*rowV)[j]=Vgap;          (*rownew)[j]=Vgap; }
            }
     rownew++;
     rowV++;
     }
}

float LAPLACE(North,West,South,East,Centre)
float North,West,South,East,Centre;
{ return (North+West+South+East - 4.0*Centre) * RFAC/4.0 + Centre;
}

int  compute(rowV,rownew,incomplete)
RP  rowV,rownew;
int  incomplete;
{ int    i,j;
 float LAPLACE();

for (i=0; i<M; i++)
{          for (j=0; i+j<M; j++)
           {
           if (j==0 & i>Y)        (*rownew)[0]=LAPLACE( (*(rowV+1))[0], (*rowV)[1],
                                                        (*(rowV-1))[0], (*rowV)[1], (*rowV)[0] );
           else if (i==0 & j>X)   (*rownew)[j]=LAPLACE( (*(rowV+1))[j], (*rowV)[j-1],
                                                        (*(rowV+1))[j], (*rowV)[j+1], (*rowV)[j] );
           else if (i>Y | j>X)    (*rownew)[j]=LAPLACE( (*(rowV+1))[j], (*rowV)[j-1],
                                                        (*(rowV-1))[j], (*rowV)[j+1], (*rowV)[j] );
           if ( fabs((*rownew)[j]-(*rowV)[j]) >= 0.0001 ) incomplete=1;
            }
rowV++;
rownew++;
}
return(incomplete);
}
```

```
main()
{
  int      i, j, repeat, iter, compute();
  GRID   V, newV;
  RP     tem, rowV, rownew;

rowV=V;
rownew=newV;
initialise(rowV,rownew);
repeat=1;
iter=1;

/* use 'repeat' for a complete run */
/*while(repeat>0)*/
while(iter<=1)
{           iter++;
            tem=rowV;
            rowV=rownew;
            rownew=tem;
            repeat=0;

            repeat=compute(rowV,rownew,repeat);
}
printf("iter = %d\n",iter-1);
}
```

Appendix (v) Part of the code of parmat.c

```
void initialise(V,newV,nproc,id)
GRID V, newV;
int   nproc, id;
{ int   i,j;
   float Vcore=1.0, Vout=0.0, Vgap=(Vcore+Vout)/2.0;
for (i=id; i<=M; i+=nproc)
           for (j=0; i+j<=M; j++)
           {   if (i+j>=M)                { V[i][j]=Vout;     newV[i][j]=Vout; }
               else if (i<=Y & j<=X)      { V[i][j]=Vcore;    newV[i][j]=Vcore; }
               else                       { V[i][j]=Vgap;     newV[i][j]=Vgap; }
           }
}

void compute(V, newV, nproc, id, incomplete, RFAC)
GRID  V, newV;
int     nproc, id, *incomplete;
float  RFAC;
{ int   i, j;
 float LAPLACE();
for (i=id; i<M; i+=nproc)
  for (j=0; i+j<M; j++)
  { if (j==0 & i>Y)          newV[i][0]=LAPLACE( RFAC, V[i+1][0], V[i][1], V[i-1][0], V[i][1], V[i][0] );
    else if (i==0 & j>X)     newV[i][j]=LAPLACE( RFAC, V[i+1][j], V[i][j-1], V[i+1][j], V[i][j+1], V[i][j] );
    else if (i>Y | j>X)      newV[i][j]=LAPLACE( RFAC, V[i+1][j], V[i][j-1], V[i-1][j], V[i][j+1], V[i][j] );
     if (fabs(newV[i][j]-V[i][j]) >= 0.0001) *incomplete=1;
  }
}

main()
{ RP      tem, rowV, rownew;
  int     i, j, *iter, *repeat, size, id, nproc;
  float RFAC=0.9;
  BARRIER *bar;

scanf("%d", &nproc);
createshate(size,rowV, rownew, repeat,iter,bar);
*repeat=1;
*iter=1;
id=spawn(nproc);
initialise(rowV,rownew,nproc,id);
barrier(bar);

/* use 'repeat' for a complete run */
/*while(*repeat>0)*/
while(*iter<=1)
      {    barrier(bar);  /* This BARRIER is vital to avoid the effect of race condition */
           if (id==0) {          *repeat=0;
                                 (*iter)++;           }
           barrier(bar);
           tem=rowV;
           rowV=rownew;
           rownew=tem;
           compute(rowV,rownew,nproc,id,repeat,RFAC);
           barrier(bar);
      }

barrier(bar);
join(nproc,id);
printf("iter=%d\n", *iter);
}
```

Appendix (vi) Part of the code of parpoint.c

```
void initialise(rowV,rownew,nproc,id)
RP rowV, rownew;
int nproc, id;
{   int  i,j;
    float Vcore=1.0, Vout=0.0, Vgap=(Vcore+Vout)/2.0;
rowV+=id; rownew+=id;
for (i=id; i<=M; i+=nproc)
 {
   for (j=0; i+j<=M; j++)
  {      if (i+j>=M)               { (*rowV)[j]=Vout;       (*rownew)[j]=Vout; }
         else if (i<=Y & j<=X)     { (*rowV)[j]=Vcore;      (*rownew)[j]=Vcore; }
         else                      { (*rowV)[j]=Vgap;       (*rownew)[j]=Vgap; }
  }
  rowV+=nproc; rownew+=nproc;
 }
}

void compute(rowV, rownew, nproc, id, incomplete, RFAC)
RP   rowV, rownew;
int   nproc, id, *incomplete;
float RFAC;
{ int   i, j;
 float LAPLACE();
rowV+=id; rownew+=id;
for (i=id; i<M; i+=nproc)
  {      for (j=0; i+j<M; j++)
         { if (j==0 & i>Y)       (*rownew)[0]=LAPLACE( RFAC, (*(rowV+1))[0], (*rowV)[1],
                                                      (*(rowV-1))[0], (*rowV)[1], (*rowV)[0] );
           else if (i==0 & j>X)  (*rownew)[j]=LAPLACE( RFAC, (*(rowV+1))[j], (*rowV)[j-1],
                                                      (*(rowV+1))[j], (*rowV)[j+1], (*rowV)[j] );
           else if (i>Y | j>X)   (*rownew)[j]=LAPLACE( RFAC, (*(rowV+1))[j], (*rowV)[j-1],
                                                      (*(rowV-1))[j], (*rowV)[j+1], (*rowV)[j] );
           if ( fabs((*rownew)[j]-(*rowV)[j]) >= 0.0001 ) *incomplete=1;
         }
  rowV+=nproc; rownew+=nproc;
  }
}

main()
{ RP      tem, rowV, rownew;
  int     i, j, *iter, *repeat, size, id, nproc;
  float   RFAC=0.9;
  BARRIER *bar;
scanf("%d", &nproc);
createshate(size,rowV, rownew, repeat,iter,bar);
*repeat=1;
*iter=1;
id=spawn(nproc);
barrier(bar);
initialise(rowV,rownew,nproc,id);
barrier(bar);
/* use 'repeat' for a complete run */
/*while(*repeat>0)*/
while(*iter<=1)
      {  barrier(bar);
         if (id==0)  {       (*iter)++;
                             *repeat=0;         }
         barrier(bar);
         tem=rowV;
         rowV=rownew;
         rownew=tem;
         compute(rowV,rownew,nproc,id,repeat,RFAC);
         barrier(bar);       }
barrier(bar);
join(nproc,id);
printf("iter=%d\n", *iter);
}
```

Appendix (vii) sisal.sis

```
define main

type GRID=ARRAY[ARRAY[REAL]];

function LAPLACE(RF,North,West,South,East,Centre: real returns real)
(North + West + South + East - 4.0 * Centre) * RF/4.0 + Centre
end function

function main(M: integer returns integer, GRID)
let     X, Y:= 6, 10;
        RFAC:=0.9; Vcore:=1.0; Vout:=0.0; Vgap:=(Vcore+Vout)/2.0;

in      for initial
        iter:=1;
        complete:=0;
        V:=     for i in 0, M
                returns array of
                   for j in 0, M-i
                   returns array of    if (i+j>=M) then Vout
                                            elseif (i<=Y & j<=X) then Vcore
                                            else Vgap
                                            end if

                   end for
                end for;

  % use 'complete' for a complete run
  %while complete=0 repeat
  while iter<=1 repeat
  iter:= old iter + 1;
  V,complete:=
        for i in 0, M
        Vij,complete:=
             for j in 0, M-i
             Vn:= if (i+j>=M) then Vout
                  elseif (j=0 & i>Y) then LAPLACE(RFAC, old V[i+1,0], old V[i,1],
                                                old V[i-1,0], old V[i,1], old V[i,0])
                  elseif (i=0 & j>X) then LAPLACE(RFAC, old V[i+1,j], old V[i,j-1],
                                                old V[i+1,j], old V[i,j+1], old V[i,j])
                  elseif (i>Y | j>X) then LAPLACE(RFAC, old V[i+1,j], old V[i,j-1],
                                                old V[i-1,j], old V[i,j+1], old V[i,j])
                  else Vcore
                  end if;
             complete:=if ABS(Vn-old V[i,j])>0.0001 then 0 else 1 end if
             returns    array of Vn
                        value of product complete
             end for
        returns array of Vij
                value of product complete
        end for;
  returns    value of iter
             value of V
  end for
end let
end function
```