Optimisation of a spline based Eulerian–Lagrangian transport solver

S. J. Leak[*] M. G. Trefry[†] F. P. Ruan[‡]

(Received 29 October 2004, revised 20 September 2005)

Abstract

Consider solute transport in saturated porous media. Eulerian–Lagrangian methods provide efficient numerical solutions with minimal numerical dispersion. We discuss the optimisation and parallelisation of a serial, spline based Eulerian–Lagrangian code (ELM2D, Fortran 90) on a 64 bit NEC SX-5 platform to support high-resolution numerical experiments. The aim was to reduce execution times by an order of magnitude whilst maintaining numerical accuracy. Profiling analysis indicated potential inefficiencies in the spline and diffusion subsystems of the code. Vectorisation of these subsystems achieved more than an order of magnitude speed increase. Use of either OpenMP or SX-5 microtasking directives was also effective in

[*]NEC Australia, Melbourne, Victoria, Australia mailto:stephen.leak@nec.com.au

[†]CSIRO Land and Water, Floreat, Western Australia, Australia mailto:mike.trefry@csiro.au

[‡]Schlumberger Petrel IS, Oslo, Norway mailto:ruan@slb.com

See http://anziamj.austms.org.au/V46/CTAC2004/Leak for this article, © Austral. Mathematical Soc. 2005. Published October 9, 2005. ISSN 1446-8735

further reducing computational expense. Benchmarking optimised solutions against 32 bit serial solutions generated on a different platform indicated good numerical agreement of overall solute distributions and plume spatial moments for strongly heterogeneous problems with $O(10^7)$ nodes. More than twenty-fold reduction in SX-5 execution times was achieved through vectorisation and parallelisation, but came at the cost of increasing memory demands. A user configurable striping parameter was introduced to the algorithm to determine the in-core storage, yielding a trade-off between execution speed and resource demand.

Contents

1 Introduction

Most potable water supplies in use around the world are subterranean. Protection of the quality of these sources is a prime human health issue, especially in locations where industrial and urban activities can readily lead to subsurface contamination. Because direct observation of groundwater movement and quality is difficult, much attention has been given to the development of theoretical methods of predicting the migration of contaminants as the groundwater moves through the heterogeneous subsurface. The problem is essentially one of advection, coupled with a local dispersion mechanism, where the carrying fluid moves through a randomly heterogeneous hydraulic conductivity field [5].

Classical methods of solving the transport equations are plagued by numerical dispersion, that is, where steep gradients in solute concentration are unable to be predicted accurately by the numerical techniques. Eulerian–Lagrangian methods (ELM) are popular tools for overcoming the numerical dispersion problem, although they can suffer from inaccuracies in the tracking and interpolation parts of the Lagrangian characteristics [1, 3]. Spline-based ELM methods have the potential to minimise these problems [8], permitting high-resolution studies of scale-dependent effects in solute transport even for large Peclet number regimes [9]. One spline-based solver is ELM2D [8], a serial Fortran 90 code designed specifically to solve the two-dimensional transport problem to high accuracy, given a divergence-free fluid velocity field as input. ELM2D uses a regular finite-difference grid and has been run successfully for problems with $\mathcal{O}(10^7)$ nodes. However, for such grid dimensions the execution times are typically large (days or weeks), effectively preventing the extension of simulations to finer grid resolutions. This work discusses the conversion of ELM2D to a vectorised multi-threaded application and the subsequent benchmarking of the new code, called ELM2D-P, against prior results.

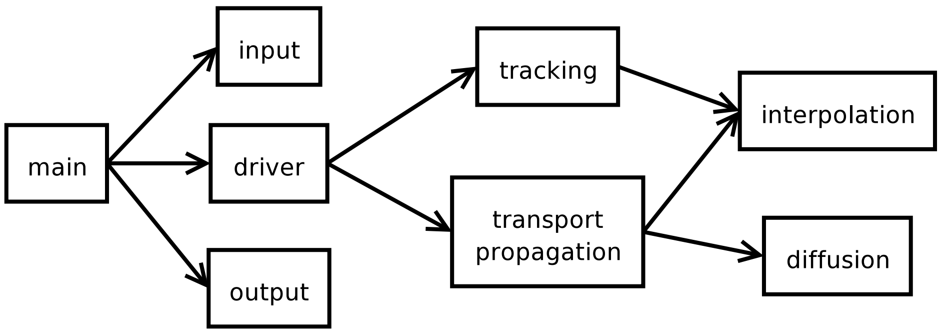


Figure 1: Code structure

2 ELM2D — Basic code structure

The general program structure is shown in Figure 1. The CPU intensive parts of the code are the tracking, interpolation and diffusion routines. Tracking involves setting up and solving an ODE at each point in the data domain; and is completed once at the beginning of every run. The transport propagation routines then propagate the data forward for a specified number of steps, using each of the interpolation and diffusion routines once at every step.

In this version of the code a taut spline function is used, and both tracking and interpolation make heavy use of two taut-spline related routines. The first, `tautsp`, describes a suitable taut spline by computing the second derivative at each point and the second, `tsvalu`, computes a selected derivative — usually the zeroth or first — at a given point on the curve. ELM2D contains support for other spline interpolators, including linear, quadratic and cubic orders. These interpolators are less attractive in terms of solution accuracy and have not been optimised, but are likely to admit similar vectorising and multi-threading efficiencies as the taut spline interpolator.

Two data sets were used; a "small grid" $(nx, ny) = (256, 256)$ domain running typically 160 propagation time steps, and a much larger (10001,1001) domain running initially 5 propagation time steps and later 500 steps. This latter "large grid" configuration represents a typical experiment for the serial code.

3 Small grid optimisations

Initial serial performance analysis with the large grids was not practical on the SX-series platform due to the CPU time required. However, previous execution of the serial code on a less expensive, slower platform (Sun Enterprise E450 with 300 MHz UltraSPARC chipset, Sun HPC Fortran 95 v6) yielded run times of more than 2 CPU weeks in 32 bit mode. Returning to the SX-series platform (NEC SX-5, FORTRAN90/SX R285), and using the small grid sets we obtain the benchmark performance information shown in Table 1.

The SX-5 [2] is a vector computer with sixteen 8 Gflop processors and 128 Gbyte of memory per node, and a recorded Linpack Rmax and Nhalf of 123 Gflops and 1340 Gflops respectively [6]. The Average vector length in Table 1 and others has an upper limit of 256 (the length of a vector register) and gives an indication of how efficiently the vector hardware is being used. The Vector operation ratio indicates how frequently the vector hardware is used. I-Cache and O-Cache miss refer to time lost due to instruction and operand (data) caches respectively; well-vectorised code makes little use of cache so this overhead is almost exclusively from scalar parts of the application. The SX-5 uses banked memory to hide memory latency, and a high number in the Bank conflict field of this or another table would suggest that memory accesses are not evenly distributed over banks. (This often occurs when loop strides are a power of two.)

In Table 1 `tautsp_` and `tsvalu_` are used for taut spline interpolations, `interp2d_tautsp_` and `vcspline2_` drive interpolation for concentration propagation and tracking respectively, `mult2_` and `findz_` are used in diffusion and `icsqrt_` is the driver for diffusion. In the case of this run, an additional 21 seconds was spent in library routines (not shown) providing pointer

Table 1: Typical serial code performance and profile for a small grid problem.

Real time	(sec)	621.79
User time	(sec)	602.61
System time	(sec)	3.16
Vector time	(sec)	36.20
Mflops		33.82
Average vector length		77.87
Vector operation ratio	(%)	26.59
Memory Size	(Mbyte)	48.03
I-Cache miss	(sec)	5.92
O-Cache miss	(sec)	116.39
Bank conflict	(sec)	0.19

Routine	Time (%)	CPU time (sec)	No. calls	Time/call (msec)
tautsp_	40.2	242.91	12109560	0.02
interp2d_tautsp_	26.1	157.84	160	986.47
tsvalu_	17.9	108.49	218807912	0.00
mult2_	3.3	19.85	320	62.05
vcspline2_	3.1	18.75	791164	0.02
findz_	2.9	17.26	480	35.96
icsqrt_	1.2	7.30	160	45.63

allocation and nullification.

The low Mflops and vector time indicate that the serial application makes almost no use of the vector processor. For optimisation a profile of where time is spent is useful; Table 1 shows the most significant routines in the profile. Armed with this analysis, we commence the optimisation of the code using inlining and vectorising techniques.

3.1 Inlining

Of the three routines comprising 80% of the total run time (`interp2d_tautsp`, `tautsp`, `tsvalu`), two have very high call counts and very short per-call times.

Vector computers can have relatively higher subroutine call overheads than scalar processors so a significant improvement may be gained by expanding such routines inline. The SX compiler was able to automatically expand `tsvalu` inline, reducing user time by 164 seconds to 438 seconds.

Note that the decrease in user time is greater than the user time originally attributed to `tsvalu`: this is due mostly to improved optimisation of the calling routine `interp2d_tautsp`. Minor differences in performance due to differences in machine load and cache use are also to be expected, but are unimportant at this stage.

3.2 Taut spline vectorisation

Vectorisation of `tautsp` was not trivial: the routine consists of a large, complex loop over `ntau` points of the curve with many data dependencies. Furthermore `ntau` is quite small, typically about 20, severely limiting the effectiveness of vectorisation. An alternative means of improving performance lies in the fact that the taut spline calculations at each point in the domain are independent, therefore vectorisation can be performed across curves rather than within them. However, this requires changes to the structure of routines calling `tautsp` as well as to `tautsp` itself.

Closer investigation of the tracking and interpolation code, described in more detail in the next two sections, reveals that about 1.6 million calls to `tautsp` were made by the tracking code and 10.5 million by the interpolation routine. The interpolation code was therefore first modified to take advantage of a vectorised version of `tautsp`.

The main challenge in vectorising `tautsp`, other than the sheer complexity of the routine, was that in many places different calculations were used depending on whether a variable was greater than, less than or equal to some given value. Vector processing must either perform all computations in all cases and mask out the unwanted results or compress the values of interest into shorter vectors for processing. Either of these reduces the efficiency at which the vector processor can operate. Fortunately, the algorithm shows considerable symmetry and with some manipulation common factors could be extracted, leaving a much more vector friendly execution path.

The key loop in the interpolation routine `interp2d_tautsp` has the following basic structure:

```
do j=1,ny
  call tautsp
end do
do i=1,nx
  do j=1,ny
    do k=-8,8
      call tsvalu
    end do
    call tautsp
    call tsvalu
  end do
end do
```

This was rearranged to call the vectorised implementation of `tautsp` (`v_tautsp`) on a collection of points simultaneously:

```
call v_tautsp(1:ny)
do point=1,nx*ny
  ! prepare arrays of independent points
end do
do k=-8,8
  do point=1,nx*ny
    call tsvalu
  end do
```

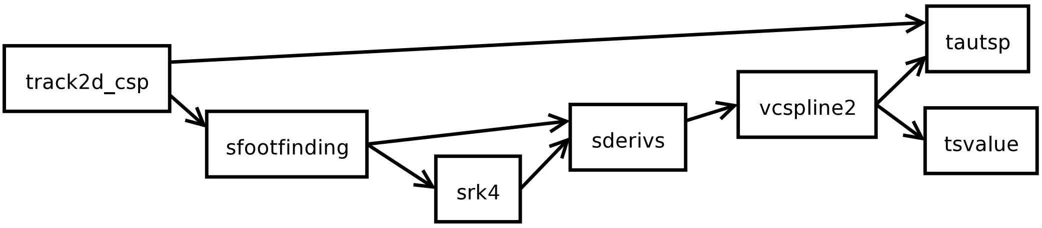


Figure 2: Structure of tracking code

```
end do
call v_tautsp(1:nx*ny)
do point=1,nx*ny
  call tsvalu
end do
```

The main effect of this is that many calls to `tautsp` are replaced by a single call to `v_tautsp`. This reduces call overhead and allows the vectorised taut spline algorithm to be used effectively. Another benefit is that `tsvalu`, which was also modified to allow efficient vectorisation of loops incorporating it, operates over a longer and more efficient vector length.

The performance after this modification was 3.8 times faster overall than before optimisation of interpolation; with the interpolation code itself running more than 30 times faster than before.

3.3 Vectorising the tracking code

Extending this optimisation to the tracking code was more complicated due to the structure spanning several routines, as depicted in Figure 2.

Each routine contributes to the algorithm as follows:

1. track2d_csp: This routine uses tautsp to set up some initial curves, then sfootfinding to perform tracking at each point in the domain.
2. sfootfinding: This routine uses a convergence loop calling srk4 in the process of solving an ODE for the current point. It is this convergence loop which most complicates the vectorised implementation of the tracking code.
3. srk4: This is a fourth-order Runge–Kutta ODE solver, see [7]. It essentially consists of a series of calls to sderivs to compute gradients at half and full step lengths.
4. sderivs, vcspline2: The algorithm for computing gradients is contained in vcspline2, for which sderivs is a wrapper. It first uses tsvalu and tautsp to set up a taut spline and then calls tsvalu again for the zeroth and first derivatives at each step.

An effect of the convergence loop in sfootfinding is that different points in the domain require a different number of calls to the lower level routines in order to converge. This was managed via a "work list" of points needing further computation; the list is created at the start of each iteration and the results copied back at the end. The "work list" approach minimises redundant calculations and uses the vector processors efficiently. This optimisation reduced the tracking time in the small test set by roughly 30 times, leaving the diffusion routines mult2 and findz as the dominant expenses in the profile.

3.4 Diffusion

Optimisation of mult2 was straightforward: removing pointer allocation and deallocation from within loops allowed the loops to vectorise. The speedup of this routine was in the order of 500 and its impact on performance was reduced to insignificance.

The algorithm for `findz` is iterative and of the form:

```
do i=2,nx*ny
  zf(i) = zf(i) - am2*zf(i-1) - am1*zf(i-ny)
end do
```

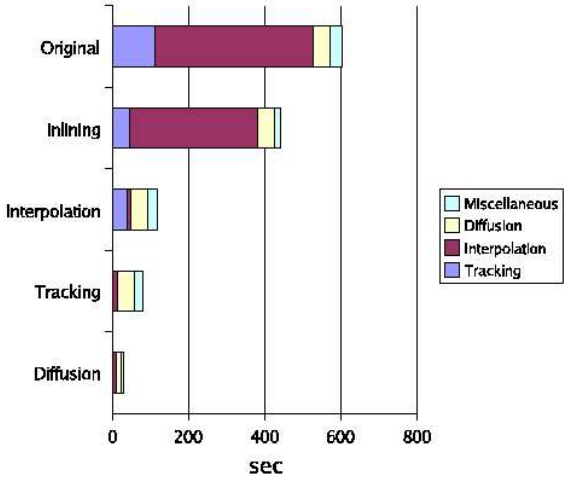
The two-dimensional nature of the algorithm (the `zf(i-ny)` part) allows a limited degree of vectorisation over `(1:ny)`. The process by which the code was rearranged is complicated, but leads to a structure of approximately:

```
! map zf(ny*nx) => w(ny,nx)
do i=2,nx
  do j=2,ny    ! partially vectorisable
    w(j,i) = w(j,i) - am2*w(j-1,i)
  end do
  do j=2,ny    ! vectorisable
    w(j,i) = w(j,i) - am1*w(j,i-1)
  end do
end do
! map w(ny,nx) => zf(ny*nx)
```

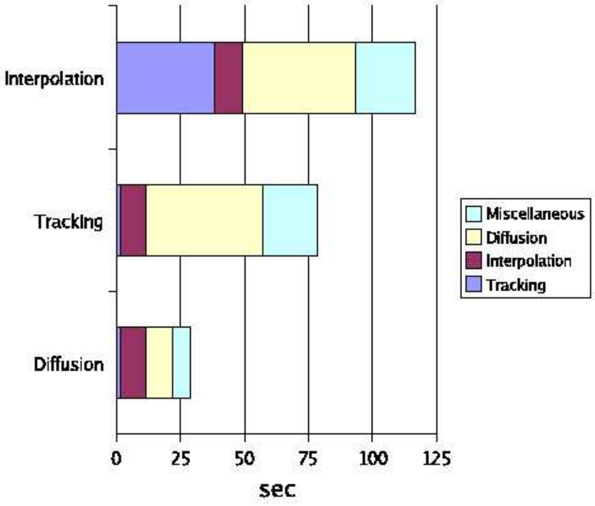
In this form the first loop can be partially vectorised with an iteration macro, and the second loop is vectorised albeit over a shorter vector length (`ny`). While the performance is mediocre it is still considerably faster than an entirely scalar implementation.

3.5 Summary

With all of these optimisations the small grid performance is approximately 20 times higher than the original code. Figure 3 shows the time spent in each



(a) All optimisations



(b) Close-up of interpolation, tracking and diffusion optimisations

FIGURE 3: Performance after various optimisations, by code section.

Table 2: Original and optimised performance of a small grid problem.

Metric		Original	Optimised
Real time	(sec)	621.79	38.57
User time	(sec)	602.61	28.46
System time	(sec)	3.16	1.03
Vector time	(sec)	36.20	16.43
Mflops		33.82	839.79
Average vector length		77.87	255.07
Vector operation ratio	(%)	26.59	97.61
Memory size	(MB)	48.03	112.03
I-Cache miss	(sec)	5.92	0.46
O-Cache miss	(sec)	116.39	1.77
Bank conflict	(sec)	0.19	1.53

section of the program after each stage of optimisation. The times for each section are determined from the top few routines in the profile, according to which parts of the code they belong. ‘Miscellaneous’ includes all other routines; its impact is relatively small and can be ignored for the moment. The optimised small grid performance is given in Table 2.

A few aspects of the optimised performance are worth noting. One is that the miscellaneous part of the program has also been improved: this is due largely to the removal of pointer allocation and deallocation from `mult2`. Another is that the vector time has actually been reduced due to more efficient use of the vector processor. Finally, the memory size has increased due to the use of work arrays storing the entire domain rather than only a single point. This last point is discussed in more detail in the next section of this report.

4 Large grid optimisations

4.1 Memory use and optimisation

With the previous optimisations a run using the large grid becomes plausible. Initially a five-step run was used, which skewed the performance heavily toward that of the tracking code but nonetheless gave an idea of the resources required for a typical experiment. The code ran at over 2.3 Gflops, used 10.7 Gbyte of main memory and ran for about 2500 seconds.

This performance is much higher than that of the small grid problem; however, memory use is also higher, at about 10.7 Gbyte. The SX-5 has 128 Gbyte of main memory so this is not excessive; however, the ability to reduce memory requirements is desirable for portability to smaller machines and to support still larger problem domains.

The largest users of memory in the vectorised code are the work arrays used, mostly by `v_tautsp`, in the tracking and interpolation parts of the program. To reduce memory use work in these parts of the program is partitioned into 'stripes' along one dimension of the domain, for example, along the ny direction. Rather than simultaneously computing at all points, the points in a single stripe are computed at each iteration. The stripe size parameter is easily configurable by the user, from ny stripes of 1 column each to 1 stripe of ny columns — that is, the entire grid in one stripe. A minimum stripe size reduced vector operation ratio marginally and vector length somewhat and increased call overhead to `v_tautsp`, consequently lowering performance slightly, but also reduced memory use for these parts of the code to about 1 Gbyte. Parts of the diffusion code use more memory than this so the overall memory use is down to about 1.7 Gbyte; this is still realistic on a modern desktop computer. Details of performance in each configuration are supplied in Table 3.

Table 3: Performance of a large grid problem over five time steps under different memory stripe size settings.

Metric		Stripe size ny	Stripe size 1
Real time	(sec)	2548.36	3484.17
User time	(sec)	2381.93	2931.20
System time	(sec)	27.45	21.16
Vector time	(sec)	2085.82	2481.66
Mflops		2328.24	1890.88
Average vector length		255.73	212.38
Vector operation ratio	(%)	99.63	99.45
Memory size	(Mbyte)	10720.03	1712.03
I-Cache miss	(sec)	4.42	15.93
O-Cache miss	(sec)	39.58	45.39
Bank conflict	(sec)	20.94	20.72

4.2 Parallelisation

Since the memory used is quite large, at least for the fastest configuration (that is, maximum stripe size), it is reasonable to use multiple processors to complete each run as quickly as possible. With this goal microtasking directives were added to a few key loops to implement a simple parallel version of the code. Initially these used the SX native directives, but were translated to use OpenMP for further portability. The parallelism gained is quite low but with a small number of CPUs the real run-time is reduced to about 2.5 hours for a 500 step experiment, from an estimated 3.5 to 4 hours in serial and over two weeks with the unoptimised code. Performance details are given in Table 4.

Note that the final, full-length parallel performance is lower than the serial performance reported in Table 3. The run profile, also summarised in the table, indicates the bottleneck: the diffusion routine `icsqrt` remains serial and unoptimised (the '$1' in the tabulated profile indicates a parallel rou-

Table 4: Performance and profile of a full length large grid run with four CPUs.

Real time	(sec)	8788.10
User time	(sec)	16494.53
System time	(sec)	146.83
Vector time	(sec)	8997.23
Mflops per CPU		932.45
Total Mflops		1816.10
Average vector length		255.29
Vector operation ratio	(%)	98.48
Memory size	(MB)	11088.00
I-Cache miss	(sec)	21.72
O-Cache miss	(sec)	1518.18
Bank conflict	(sec)	94.08

Routine	Tasks	Time (%)	Time (sec)
icsqrt_	1	51.5	3949.70
v_tautsp$1_	4	11.2	861.29
interp2d_tautsp$1_	4	7.7	594.38
v_findz$1_	4	4.5	343.65

tine). This routine forms part of an incomplete Cholesky decomposition and contains strong dependencies, making both vectorisation and parallelisation very difficult. This is a well known characteristic of incomplete Cholesky factorisers, and much attention has been given elsewhere to optimising them for computation. At this time the overall performance gain is satisfactory so improvements to icsqrt will not be attempted.

5 Accuracy: ELM2D-P versus ELM2D

The accuracy of ELM2D-P is best assessed in comparison with the benchmark code base of ELM2D. As an aside, we note that simple porting of code from platform to platform is often sufficient to induce differences in output for complicated numerical codes. Therefore we may anticipate differences in the output solutions between the 64 bit NEC SX-5 executable and the 32 bit Sun E450 executable.

A variety of small and large grid problems were used to test the accuracy of ELM2D-P; here we give results from one large grid problem for a moderately heterogeneous hydraulic conductivity field. Figure 4 shows windowed solute plumes calculated by ELM2D-P and ELM2D from the same input fluid velocity field with an initial distribution of a thin rectangular source (with long axis aligned vertically) at the left of the grid and mean fluid flow from left to right. The solute plumes are rendered with a logarithmic concentration scale. The two plumes are closely similar, with only a few subtle differences visible in the figure.

A more quantitative analysis is given by calculating relative (percentage) discrepancies between plume spatial moments and peak concentrations for the two solutions. Figure 5 shows how the relative discrepancies between the models evolve with time. The discrepancies were always less than 0.5% for all the measures tested, which is within the bounds of differences observed after compiling and executing the Sun ELM2D code base unchanged on the NEC SX-5 (grey curve, M_{20} small grid). Interestingly, porting the 64 bit ELM2D-P code to a Sun platform results in output solutions that are (bitwise) identical to the 64 bit NEC SX-5 solutions. This suggests that the ELM2D base algorithm is prone to minor inaccuracies on 32 bit architectures, but such inaccuracies in ELM2D-P are reduced or eliminated on 64 bit platforms.

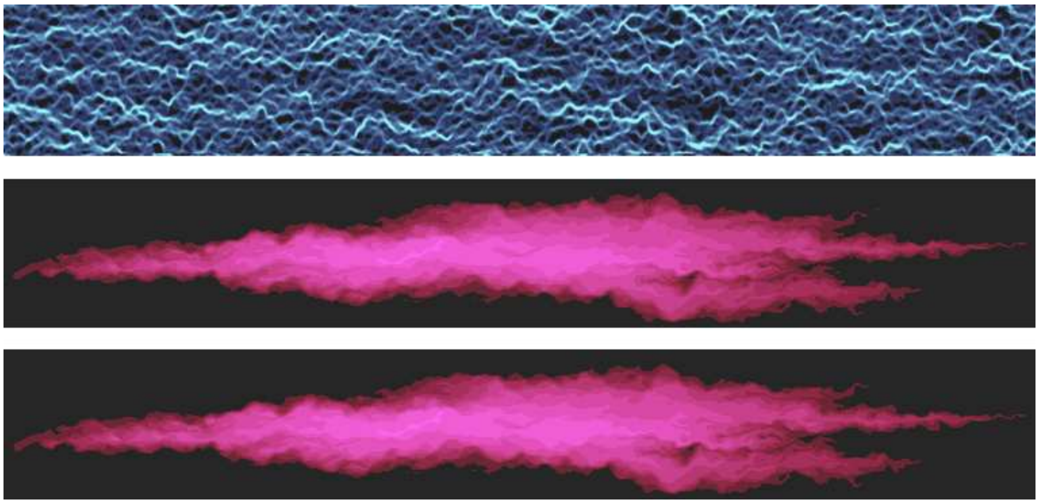


Figure 4: Solute plumes calculated for a large grid problem with elm2d-p (middle plot, 64 bit nec sx-5) and elm2d (bottom plot, 32 bit Sun e450) after 100 time steps. The magnitude of the input fluid velocity field is also shown (top plot).

6 Final remarks and further work

The optimisation of elm2d has been remarkably successful, with the possibility of one or two extra orders of magnitude in grid size now well within reach. The most obvious next step is the optimisation of the incomplete Cholesky code: this code block accounts for 50% of the run time for a typical large problem. Recent parallel factorisations may be relevant here [4]. Finally, we note that extension to three spatial dimensions is a crucial and required step for the most accurate modelling of dispersive transport in porous media.

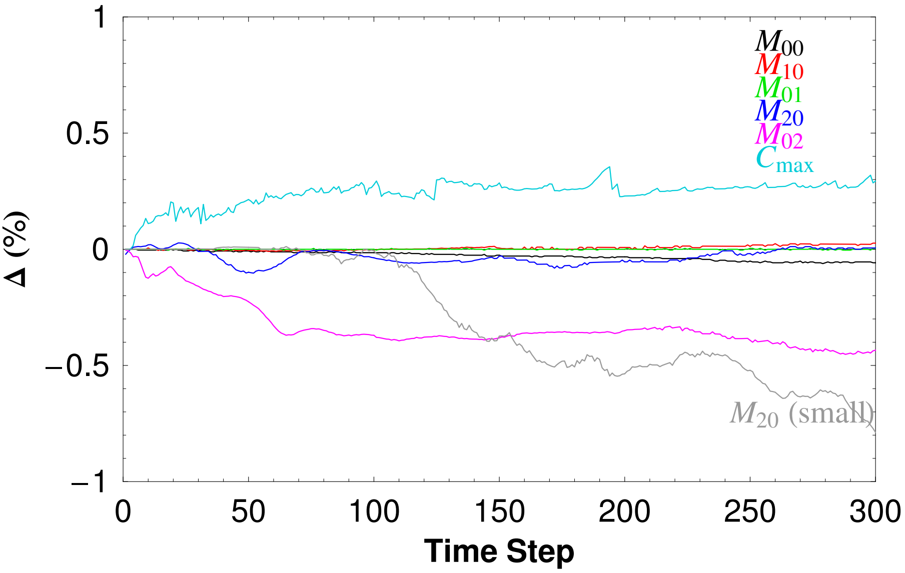


Figure 5: Percentage differences between ELM2D and ELM2D-P for a large grid problem. Differences are calculated for plume spatial moments M_{ij} and the peak plume concentration $C_{\max}$.

Acknowledgment: This work was performed with the assistance of the High Performance Computing and Communications Centre, Melbourne, Australia.

References

[1] Bierbrauer, F., Soh, W. K. and Yuen, W. Y. D., On some developments and evaluation of an Eulerian–Lagrangian method for the transport equation *ANZIAM J.* 42(E):C238–C262, 2000.
http://anziamj.austms.org.au/V42/CTAC99/Bier C1037

[2] Kinoshita, K., Hardware System of the SX Series *NEC Res. and Develop.* 39(4):362–368, 1998. C1039

[3] de Oliviera, A. and Baptista, A. M., A comparison of integration and interpolation Eulerian-Lagrangian methods *Int. J. Numer. Methods Fluids* 21, 183–204, 1995. C1037

[4] Magolu monga Made, M. and van der Vorst, H. A., ParIC: A family of parallel Incomplete Cholesky Preconditioners HPCN Europe, 89–98, 2000. [Online] http://citeseer.ist.psu.edu/made00paric.html. C1052

[5] McLaughlin, D. and Ruan, F., Macrodispersivity and large-scale hydrogeologic variability *Transp. Porous Media* 42(1):133–154, 2001. C1037

[6] Meuer, H., Strohmaier, E., Dongarra, J. and Simon, H., Top500 Supercomputer Sites [Online] http://www.top500.org. 2005. C1039

[7] Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P., Numerical Recipes in Fortran 90: The art of parallel scientific computing Volume 2 of Fortran Numerical Recipes, Cambridge University Press, Cambridge, UK, 1996. http://www.nr.com/ C1044

[8] Ruan, F. and McLaughlin, D., An investigation of Eulerian–Lagrangian methods for solving heterogeneous advection-dominated transport problems *Water Resour. Res.* 35(8):2359–2373, 1999. C1037

[9] Trefry, M. G., Ruan, F. P. and McLaughlin, D., Numerical simulations of preasymptotic transport in heterogeneous porous media: Departures from the Gaussian limit *Water Resour. Res.* 39(3), Article 1063, 2003. http://dx.doi.org/10.1029/2001WR001101 C1037