High Performance Computations for Large Scale Simulations of Subsurface Multiphase Fluid and Heat Flow

Erik Elmroth Chris Ding Yu-Shu Wu
Lawrence Berkeley National Laboratory,
University of California, Berkeley, CA

Abstract

TOUGH2 is a widely used reservoir simulator for solving subsurface flow related problems such as nuclear waste geologic isolation, environmental remediation of soil and groundwater contamination, and geothermal reservoir engineering. It solves a set of coupled mass and energy balance equations using a finite volume method. This contribution presents a parallel version of TOUGH2. The parallel implementation first partitions the unstructured computational domain. For each time step, a set of coupled non-linear equations is solved with Newton iteration. In each Newton step, a Jacobian matrix is calculated and an ill-conditioned non-symmetric linear system is solved using a preconditioned iterative solver. Communication is required for convergence tests and data exchange across partitioning borders. Parallel performance results on Cray T3E-900 are presented for two real application problems arising in the Yucca Mountain nuclear waste site study. The execution time is reduced from 7504 seconds on two processors to 126 seconds on 128 processors for a 2D problem involving 52,752 equations. For a larger 3D problem with 293,928 equations the time decreases from 10055 seconds on 16 processors to 329 seconds on 512 processors.

Keywords. Groundwater flow, grid partitioning, iterative linear solvers, preconditioners.

1 Introduction

Subsurface flow related problems touch many important areas in today's society, such as natural resource development, nuclear waste underground storage, environmental remediation of groundwater contamination, and geothermal reservoir engineering. Because of the complexity of model domains and physical processes involved, numerical simulation play vital roles in the solutions of these problems.

This contribution describes a parallel implementation of the widely used TOUGH2 software package [6, 7] for numerical simulation of flow and transport in porous and fractured media. The contribution includes descriptions of algorithms and methods used in the parallel implementation and performance evaluation for parallel simulations with up to 512 processors on a Cray T3E-900 on two real application problems.

The serial version of TOUGH2 (Transport Of Unsaturated Groundwater and Heat version 2) is now being used by over 150 organizations in more than 20 countries (see [8] for some examples). The major application areas include geothermal reservoir simulation, environmental remediation, and nuclear waste isolation. TOUGH2 is one of the official codes used in the US Department of Energy's civilian nuclear waste management for the evaluation of the Yucca Mountain site as a repository for nuclear wastes. In this context arises the largest and most demanding applications for TOUGH2 so far. Scientists at Lawrence Berkeley National Laboratory are currently developing a 3D flow model of the Yucca Mountain site, involving computational grids of 10^5 to 10^6 grid blocks, and related coupled equations of water and gas flow, heat transfer and radionuclide migration in subsurface [1]. Considerably larger and more difficult applications are anticipated in the near future, with the analysis of solute transport, with ever increasing demands on spatial resolution and a comprehensive description of complex geological, physical and chemical processes. High performance capability of the TOUGH2 code is essential for these applications.

Some early results from this project was presented in [2].

2 The TOUGH2 Simulation

The TOUGH2 simulation package solves mass and energy balance equations that describe fluid and heat flow in general multiphase, multicomponent systems. The fundamental balance equations have the following form:

$$\frac{d}{dt}\int_V M^{(k)} dV = \int_S \mathbf{F}^{(k)} \cdot \mathbf{n} dS + \int_V q^{(k)} dV,$$

where the integration is over an arbitrary volume V, which is bounded by the surface S. Here $M^{(k)}$ denotes mass for the k-th component, (water, gas, heat, etc), $\mathbf{F}^{(k)}$ is the flux of fluids and heat through the surface, and $q^{(k)}$ is source or sink inside V. This is a general form. All flow and mass parameters can be arbitrary non-linear functions of the primary thermodynamic variables, such as density, pressure, saturation, etc.

Given a computational geometry, space is discretized into many small volume blocks. The integral on each block becomes a variable; this leads naturally to the finite volume method, resulting in the following ordinary differential equations:

$$\frac{dM_n^{(k)}}{dt} = \frac{1}{V_n} \sum_m A_{nm} F_{nm}^{(k)} + q_n^{(k)},$$

where V_n is the volume of the block n, and A_{nm} is the interface area bordering between blocks n, m and F_{nm} is the flow between them. Note that flow terms usually contain spatial derivatives, which are replaced by simple difference between variables defined on blocks n, m and divided by the distances between the block centers. See Figure 1 for an illustration.

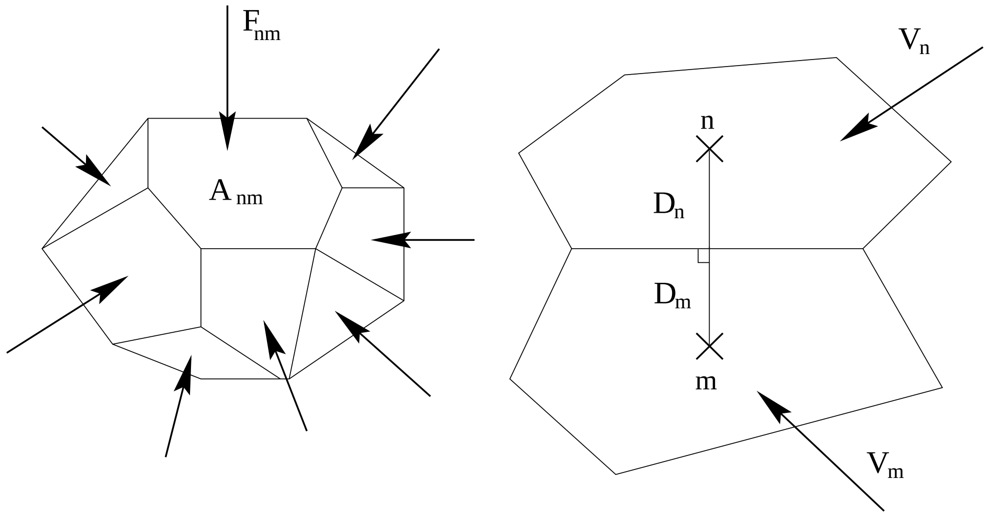


Figure 1: Space discretization and geometry data.

Initialization and setup
do *Time step advance*
  do *Newton iteration*
    Calculate the Jacobian matrix
    Solve linear system
  end do
end do
output

Figure 2: Sketch of main loops for the TOUGH2 simulation.

Time is implicitly discretized as a first order difference equation:

$$M_n^{(k)}(x^{(t+1)}) - M_n^{(k)}(x^{(t)}) = \frac{\Delta t}{V_n}(\sum_m A_{nm} F_{nm}^{(k)}(x^{(t+1)}) + V_n q_n^{(k)}),$$

where the vector $x^{(t)}$ consists of prime variables at time t. Flow and source/sink terms on the right hand side are evaluated at $t + \Delta t$ for numerical stability for the multi-phase problems. This lead to coupled nonlinear algebraic equations, which are solved using Newton's method.

3 Computational Procedure

The main solution procedures can be schematically outlined as in Figure 2.

After reading data and setting up the problem, the time consuming parts are the main loops for time stepping, Newton iteration, and the iterative linear solver. At each time step, the nonlinear discretized coupled algebraic equations are solved with the Newton method. Within each Newton iteration, the Jacobian matrix is first calculated by numerical differentiation. The implicit system of linear equations is then solved using a sparse linear solver with preconditioning. After several Newton iterations, the convergence is checked by a control parameter, which measures the maximum component of the residual in the Newton iterations. If the Newton iterations converge, the time will advance one more time step, and the process repeats until the pre-defined total time is reached.

If the Newton procedure does not converge after a preset max-Newton-iteration, the current time step is reduced (usually by half) and the Newton procedure is tried for the reduced time step. If converged, the time will advance; otherwise, time step is further reduced and another round of Newton iteration follows. This procedure is repeated until convergence in the Newton step is reached.

The system of linear equation is usually very ill-conditioned, and requires very robust solvers. The dynamically adjusted time step size is the key to overcome the combination of possible convergence problems for the Newton iteration and the linear solver. For this highly dynamic system, the trajectory is very sensitive to variations in the convergence parameters.

Computationally, the major part (about 65%) of the execution time is spent on solving the linear systems, and the second major part (about 30%) is the assembly of the Jacobian matrix.

4 Designing the Parallel Implementation

The aim of this work is to develop a parallel prototype of TOUGH2, and to demonstrate its ability to efficiently solve problems significantly larger than problems that have previously been solved using the serial version of the software. The problems should be larger both in the number of blocks and the number of equations per block. The target computer system for this prototype version of the parallel TOUGH2 is the 696 processor Cray T3E-900 at NERSC, Lawrence Berkeley National Laboratory.

In the following sections, we give an overview of the design of the main steps, including grid partitioning, grid block reordering, assembly of the Jacobian matrix, and solving the linear system, as well as some further details about the parallel implementation.

4.1 Grid Partitioning and Grid Block Reordering

Given a finite domain as described in Section 2, we will in the following consider the dual mesh (or grid), obtained by representing each *block* (or volume element) by its centroid and by representing the interfaces between blocks by *connections*. (The nomenclature blocks and connections is used in consistency with the original TOUGH2 documentation [7].) The physical properties for blocks and their interfaces are represented by data associated with the blocks and connections, respectively.

In TOUGH2 the computational domain is defined by the set of all connections given as input data. From this information, an adjacency matrix is constructed, i.e., a matrix with a non-zero entry for each element (i, j) where there is a connection between blocks i and j. In the current implementation the value 1 is always used for non-zero elements, but different weights may be used. The adjacency matrix is stored in a compressed row format, called CSR format, which is a slight modification of the Harwell-Boeing format.

The actual partitioning of the grid into p almost equal-sized parts is performed using three different partitioning algorithms, implemented in the METIS software package version 4.0 [5]. The three algorithms are here denoted the *K-way*, the *VK-way* and the *Recursive* partitioning algorithm, in consistency with the METIS documentation.

K-way is multilevel version of a traditional graph partitioning algorithm that minimizes the number of edges that straddle the partitions. *VK-way* is a modification of *K-way* that instead minimizes the actual total communication volume. *Recursive* is a recursive bisection algorithm which objective is to minimize the number of edges cut.

After partitioning the grid on the processors, the blocks (or more specifically, the vector elements and matrix rows associated with the blocks) are reordered by each processor to a local ordering. The blocks for which a processor computes the results are denoted the *update* set of that processor. The update set can be further partitioned into the *internal* set and the *border* set. The border set consists of blocks with an edge to a block assigned to another processor and the internal set consists of all other blocks in the update set. Blocks not included in the update set but needed (read only) during the computations defines the *external* set.

Figure 3 illustrates how the blocks can be distributed over the processors. (The vertices of the graph represent blocks and the edges represent connections.) Table 1 shows how the blocks are classified in the update and the external sets and how the update sets are further divided into internal and border sets.

Table 1: Example of block distribution and local ordering for the Internal, Border, and External sets. The elements are here placed in local order and the global numbering illustrates the reordering.

	(*Internal* | *Border* || *External*)
Processor 0:	$(7, 11 \mid 8, 12 \parallel 1, 9, 13)$
Processor 1:	$(2, 3 \mid 1, 4, 9 \parallel 8, 12, 5, 10)$
Processor 2:	$(6, 14 \mid 5, 10, 13 \parallel 12, 4, 9)$

In order to facilitate the communication of elements corresponding to border/external blocks, the local renumbering of the nodes is made in a particular way. All blocks in the update set precede the blocks in the external set, and in the update set, all internal blocks

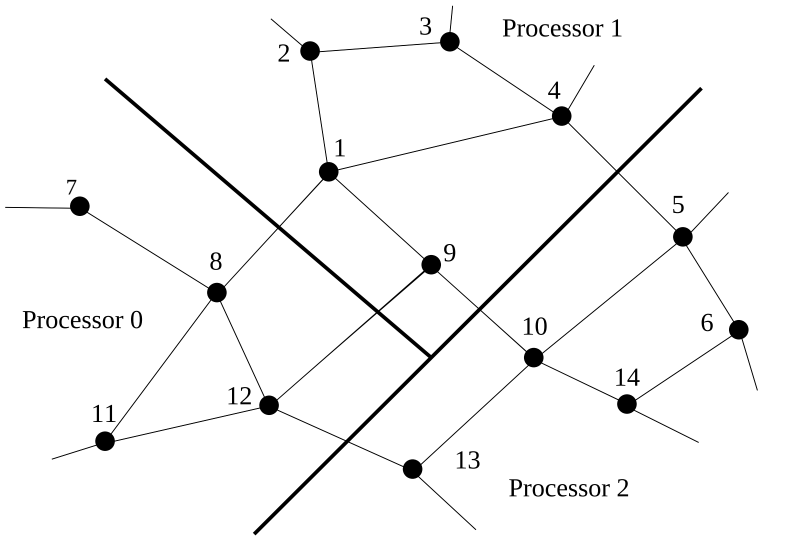


Figure 3: A grid partitioning on 3 processors.

precede the border blocks. Finally, the external blocks are ordered internally with blocks assigned to a specific processor placed consecutively. One possible ordering is given as an example in Table 1.

The consecutive ordering of the external blocks that reside on each processor makes it possible to receive data corresponding to these blocks into appropriate vectors without use of buffers and with no need for further reordering, provided that the sending processor has access to the ordering information. However, it is not possible in general to order the border blocks so that transformations can be avoided when sending, basically because some blocks in the border set may have to be sent to more than one processor.

4.2 Jacobian Matrix Calculations

A new Jacobian matrix is calculated once for each Newton step, i.e., several times for each Time step of the algorithm. In the parallel algorithm, each processor is responsible for computing the rows of the Jacobian matrix that correspond to blocks in the processor's *update* set. All derivatives are computed numerically.

The Jacobian matrix is stored in the Distributed Variable Block Row format (DVBR) [4]. All matrix blocks are stored row wise, with the diagonal blocks stored first in each block row. The scalar elements of each matrix block are stored in column major order. The use of dense matrix blocks enable use of dense linear algebra software, e.g., optimized level 2 (and level 3) BLAS for subproblems. The DVBR format also allows for a variable number of equations per block.

Computation of the elements in the Jacobian matrix is basically performed in two phases.

The first phase consists of computations relating to individual blocks. At the beginning of this phase, each processor already holds the information necessary to perform these calculations. The second phase include all computations relating to interface quantities, i.e., calculations using variables corresponding to pairs of blocks. Before performing these computations, exchange of relevant variables is required. For a number of variables, each processor sends elements corresponding to *border* blocks to appropriate processors, and it receives elements corresponding to *external* blocks.

4.3 Linear Systems

The non-symmetric linear systems to be solved are generally very ill-conditioned and difficult to solve. Therefore, the parallel implementation of TOUGH2 is made so that different iterative solvers and preconditioners easily can be tested. All results presented here have been obtained using the stabilized bi-conjugate gradient method (BICGSTAB) [11] in the Aztec software package [4], with 3×3 Block Jacobi scaling and a domain decomposition based preconditioner with possibly overlapping subdomains, i.e., Additive Schwarz (see, e.g., [10]), using the ILUT [9] incomplete LU factorization.

The domain decomposition based procedure can be performed with different levels of overlapping, and for the case overlap $= 0$ the procedure turns into another variant of Block Jacobi preconditioner. In order to distinguish the 3×3 Block Jacobi scaling from the full subdomain Block Jacobi scaling obtained by chosing overlap $= 0$ in the domain decomposition preconditioning procedure, we will refer to the former as the Block Jacobi *scaling* and the latter as the domain decomposition based *preconditioner*, though both are of course preconditioners.

As an illustration of the difficulties arising in these linear systems, we would like to mention a very small problem from the Yucca Mountain simulations mentioned in the Introduction. This non-symmetric problem includes 45 blocks, 3 equations per block, and 64 connections. When solving the linear system, the Jacobian matrix is of size 135×135 with 1557 nonzero elements. For the first Jacobian generated (in the first Newton step of the first Time step), i.e., the matrix involved in the first linear system to be solved, the largest and smallest singular values are 2.48×10^{32} and 2.27×10^{-12}, respectively, giving the condition number 1.1×10^{44}.

By applying block Jacobi scaling, where each block row is multiplied by the inverse of its 3×3 diagonal block, the condition number is significantly reduced. The scaling reduces the largest singular value to 7.69×10^{3} and the smallest is increased to 9.83×10^{-5}, altogether reducing the condition number to 7.8×10^{7}. This is, however, still an ill-conditioned problem. Therefore, the domain decomposition based preconditioner with incomplete LU factorization mentioned above is applied after the block Jacobi scaling. This procedure has shown to be absolutely vital for convergence on problems that are significantly larger.

4.4 Parallel Implementation

In this section, we outline the parallel implementation by describing the major steps in some important routines. In all, the parallel TOUGH2 includes about 20,000 lines of Fortran code (excluding the METIS and Aztec packages) in numerous subroutines using MPI for message passing [3]. However, in order to understand the main issues in the parallel implementation, it is sufficient to focus on a couple of routines. Of course, several other routines are also modified compared to the serial version of the software, but these details would only be distracting.

Cycit

Initially, processor 0 reads all data describing the problem to be solved, essentially in the same way as in the serial version of the software. Then, all processors call the routine *Cycit* which contains the main loops for time stepping and Newton iterations. This routine also initiates the grid partitioning and data distribution. The partitioning described in Section 4.1 defines how the input data should be distributed on the processors. The distribution is performed in several routines called from *Cycit*.

There are five categories of data to be distributed and possibly reordered. Vectors with elements corresponding to grid blocks are distributed according to the grid partitioning and reordered to the local order with *Internal*, *Border*, and *External* elements as described in Section 4.1. Vectors with elements corresponding to connections are distributed and adjusted to the local grid block numbering after each processor have determined which connections are involved in its own local partition. Vectors with elements corresponding to sinks and sources are replicated in full before each processor extracts and reorders the parts needed. There are in addition a number of scalars and small vectors and matrices that are fully replicated, i.e., data structures which sizes do not depend on the number of grid blocks or connections. Finally, processor 0 constructs the data structure for storing the Jacobian matrix and distributes appropriate parts to the other processors. This include all integer vectors defining the matrix structure but not the large array for holding the floating point numbers for the matrix elements.

As the problem is distributed, the time stepping procedure begins. A very brief description of the routine *Cycit* is given in Figure 4. In this description, lots of details have been omitted for clarity, and calls have been included to a couple of routines that require further description.

ExchangeExternal

The routine ExchangeExternal is of particular interest for the parallel implementation. The main loop of this routine is outlined in Figure 5. When called by all processors with a vector and a scalar *noel* as arguments, an exchange of vector elements corresponding to external grid blocks is performed between all neighboring processors. The parameter *noel* is the number of vector elements exchanged per external grid block. Some additional parameters that defines

```
Cycit(...)

Initialization, grid partitioning, data distribution etc
Set up first time step and the first Newton step
num_sec_par = number of secondary variables (in PAR) per grid block
Iter = 0
Time = StartTime
while Time < EndTime
    Newton_converged = false
    while not Newton_converged
        Iter = Iter + 1
        call Multi(...)
        Newton_converged = result from convergence test
        if Newton_converged
            Update primary variables
            Increment Time, define new time step and set Iter = 0
        else
            if Iter ≤ MaxIter
                Solve linear system
                call Eos(...)
                call ExchangeExternal(PAR, num_sec_vars)
            end if
            if Iter > MaxIter or Physical properties out of range
                if time step has been decreased too many times
                    Stop execution
                    Print message about failure to solve problem
                else
                    Reduce time step
                    call Eos(...)
                    call ExchangeExternal(PAR, num_sec_vars)
                    Iter = 0
                end if
            end if
        end if
    end while
end while
```

Figure 4: Outline of the routine *Cycit*, executed by all processors.

```
ExchangeExternal(vector, noel)

recvstart = noel*(N_internal + N_border) + 1
tag = 1000
iw = 1

do i = 1, num_neighbors
    proc = neighbors(i)

    rlen = noel*recvlength(i)
    call MPI_IRECV(vector(recvstart), rlen, MPI_DOUBLE_PRECISION,
        proc, tag+myid, MPI_COMM_WORLD, req(2*i-1), ierr)
    recvstart = recvstart + rlen

    slen = noel*sendlength(i)
    call pack(i, vector, sendindex, slen, work(iw), noel)
    call MPI_ISEND(work(iw), slen, MPI_DOUBLE_PRECISION,
        proc, tag+proc, MPI_COMM_WORLD, req(2*i), ierr)
    iw = iw + slen
end do
call MPI_WAITALL(2*num_neighbors, req, stat, ierr)
```

Figure 5: Outline of the routine ExchangeExternal. When simultaneously called by all processors it performs an exchange of *noel* elements per external grid block for the data in *vector*.

the current partition, e.g., information about neighbors etc, need also to be passed to the routine, but we have for clarity chosen not to include them in the figure. Though some details are omitted, we have chosen to include the full MPI syntax (using Fortran interface) for the communication primitives. The routine *pack*, called by *ExchangeExternal*, copies appropriate elements from *vector* into a consecutive *work* array. The external elements for a given processor are specified by *sendindex*.

We remark that the elements can be stored directly into the appropriate vector when received (since external blocks are ordered consecutively for each neighbor), whereas the border elements to be sent need to be packed into a consecutive work space before they are sent.

Note that we use the nonblocking MPI routines for sending and receiving data. With use of blocking routines we would have had to assure that all messages are sent and received in an appropriate order to avoid deadlock. When using nonblocking primitives, the sends and receives can be made in arbitrary order. A minor inconvenience with use of the nonblocking routines is that the work space used to store elements to be sent need to be large enough to store all elements a processor is to send to all its neighbors.

Multi

The routine *Multi* is called to set up the linear system, i.e., the main part of the computations in *Multi* is for computing the elements of the Jacobian matrix. Computationally, *Multi* performs three major steps. First it performs all computations that depend on individual grid blocks. This is followed by computations of terms arising from sinks and sources.

So far all computations can be made independently by all processors. The last computational step in *Multi* is for interface quantities, i.e., computations involving pairs of grid blocks. Before performing this last step, and exchange of external variables is required for the vectors X (primary variables), DX (the last increments in the Newton process), $DELX$ (small increments of the X values, used to calculate incremental parameters needed for the numerical calculation of the derivatives), and R (the residual). The number of elements to be sent per external grid block equals the number of equations per grid block, for all four vectors. This operation is performed by calling *ExchangeExternal* before performing the computations involving interface quantities.

Eos3 and other Eos routines

The thermophysical properties of fluid mixtures needed in assembling the governing mass and energy balance equations are provided by a routine called Eos (Equations of state). The main task for the Eos routine is to provide values for all secondary (thermophysical) variables as functions of the primary variables, though it also performs some additional important tasks (see [7], pages 17–26 for details).

Several Eos routines are available for TOUGH2, and new Eos routines will become available. However, Eos3 is the only one that have been used in this parallel implementation. In order to provide maximum flexibility, we strive to minimize the amount of changes that needs to be done to the Eos routine when moving from the serial to the parallel implementation. This have been done by organizing data and assigning appropriate values to certain variables before calling the Eos routine. In the current parallel implementation, the Eos3 routine from the serial code can be used unmodified, with the exception of some write statements. Though, this still needs to be verified in practice, we believe that the current parallel version of TOUGH2 can handle also other Eos routines, with the only exception being some write statements needing adjustments.

5 Performance Analysis

Parallel performance evaluation have been performed for a 2D and a 3D real application problem arising in the Yucca Mountain nuclear waste site study. Results have been obtained for up to 512 processors of the Cray T3E-900 at NERSC, Lawrence Berkeley National Laboratory.

The linear systems have been solved using BICGSTAB with 3×3 Block Jacobi scaling and a domain decomposition based preconditioner with the ILUT incomplete LU factorization. Different levels of overlapping have been tried for this procedure, though all results presented are for non-overlapping tests, which in general have shown to give good performance. The stopping criteria used for the linear solver is $\frac{\|r\|_2}{\|b\|_2} \leq 10^{-4}$, where r and b denote the residual and the right hand side, respectively.

Both test problems require simulated times of 10^4 to 10^5 years, which would require a significant execution time also with good parallel performance and a large number of processors. In order to investigate the parallel performance, we have therefore limited the simulated time to 10 years for the 2D problem and 0.1 year for the 3D problem, which still require enough time steps to perform the analysis of the parallel performance. A shorter simulated time will of course give the initialization phase unproportionally large impact on the performance figures. The initialization phase is therefore excluded from the timings.

Tests have been performed using the *K-way*, the *VK-way*, and the *Recursive* partitioning algorithms in METIS. As we will see later, different orderings of the grid blocks lead to variations in the time discretization following from the unstructured nature of the problem. This in turn lead to variations in the number of time steps required and thereby in the total amount of work performed. By trying all three partitioning algorithms and chosing the one that leads to the best performance for each problem and number of processors, we reduce these somewhat "artificial" performance variations resulting from differences in the number of time steps required. For all results presented, we indicate which partitioning algorithm have been used.

5.1 Results for 2D and 3D Real Application Problems

The 2D problem consists of 17,584 blocks, 3 components per block and 43,815 connections between blocks, giving in total 52,752 equations. The Jacobian matrix in the linear systems to be solved for each Newton step is of size 52,752 $\times$ 52,752 with 946,926 non-zero elements.

The topmost graph in Figure 6 illustrates the reduction in execution time for increasing number of processors. The execution time is reduced from 7504 seconds (i.e., 2 hours, 5 minutes, and 4 seconds) on two processors to 126 seconds (i.e., 2 minutes and 6 seconds) on 128 processors.

The parallel speedup for the 2D problem is presented in the second graph of Figure 6. Since the problem cannot be solved on one processor with the parallel code the speedup is normalized to be 2 on two processors, i.e., the speedup on p processors is calculated as $2T_2/T_p$, where T_2 and T_p denote the wall clock execution time on 2 and p processors, respectively. For completeness we also report that the execution time for the original serial code is 8245 seconds on the 2D problem.

The 3D problem consists of 97,976 blocks, 3 components per block and 396,770 connections between blocks, giving in total 293,928 equations. The Jacobian matrix in the linear systems to be solved for each Newton step is of size 293,928 $\times$ 293,928 with 8,023,644 non-zero elements.

The topmost graph in Figure 7 illustrates the reduction in execution time for the 3D problem for increasing number of processors. Memory and batch system time limits prohibits tests on less than 16 processor. Results are therefore presented for 16, 32, 64, 128, 256, and 512 processors. The execution time is significantly reduced as the number of processors is increased, all the way up to 512 processors. It is reduced from 10055 seconds (i.e., 2 hours, 47 minutes, and 35 seconds) on 16 processors to 329 seconds (i.e., 5 minutes and 29 seconds) on 512 processors. The ability to efficiently use larger number of processors is even better illustrated by the speedup shown in the second graph of the Figure 7. The speedup is defined as $16T_{16}/T_p$ since performance result are not available for smaller number of processors.

The results clearly demonstrate very good parallel performance up to very large number of processors for both problems. We observe speedups up to 119.1 on 128 processors for the 2D problem and up to 489.3 on 512 processors for the 3D problem.

When repeatedly doubling the number of processors from 2 to 4, from 4 to 8, etc, up to 128 processors for the 2D problem, we obtain the speedup factors 1.58, 2.85, 2.19, 1.91, 1.96, and 1.62. For the 3D problem, the corresponding speedup factors when repeatedly doubling the number of processors from 16 to 512 processors are 2.70, 2.28, 1.95, 1.50, and 1.69.

As 2.00 would be the ideal speedup each time the number of processors is doubled, the speedup, e.g., 2.70 and 2.28 for the 3D problem are often called super-linear speedup. We will present the explanations for this in later sections.

Overall the parallel performance is very satisfactory, and we complete this analysis by providing some insights and explaining the super-linear speedup.

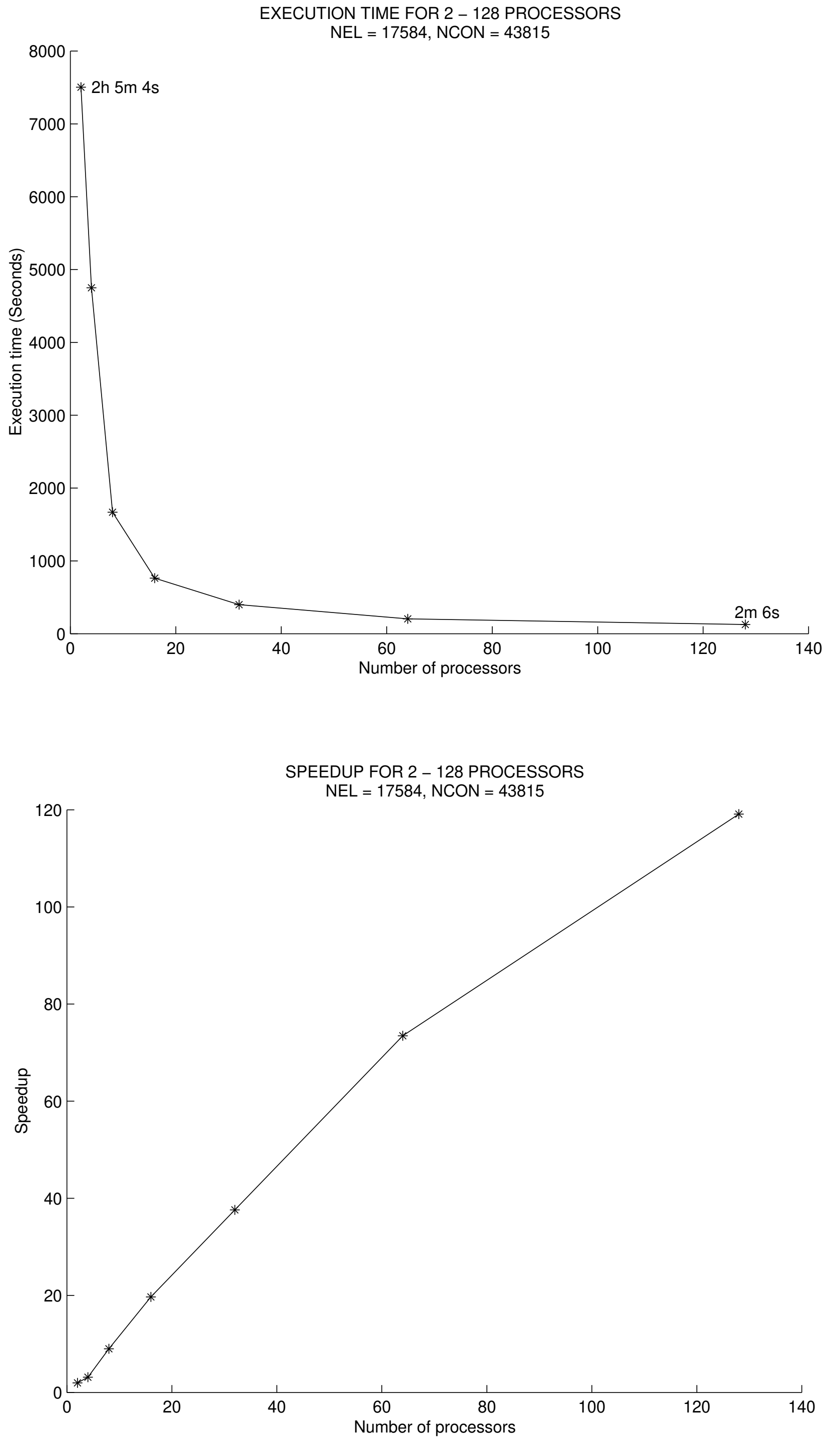
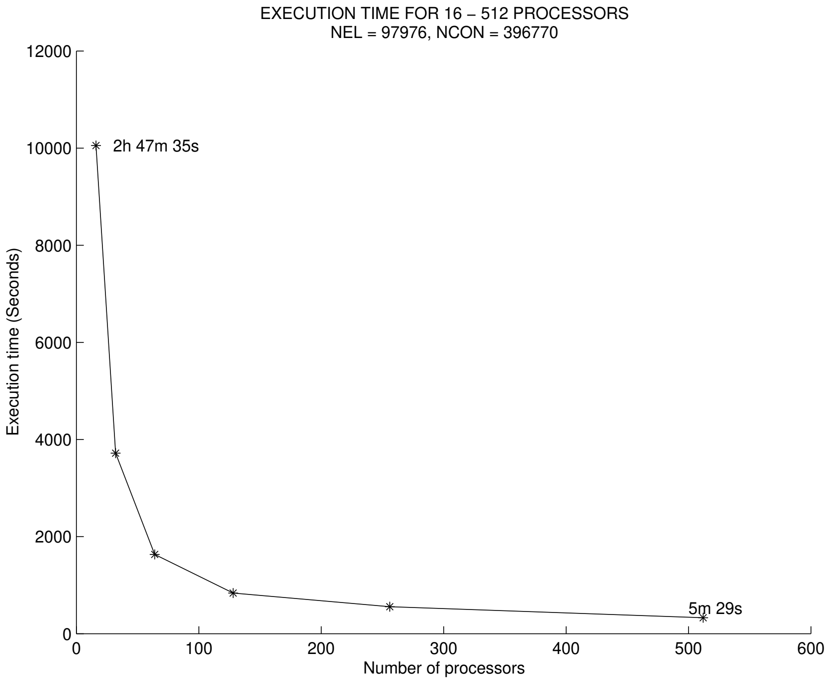


Figure 6: Execution time and parallel speedup on the 2D problem for 2, 4, 8, 16, 32, 64, and 128 processors on the Cray T3E-900.



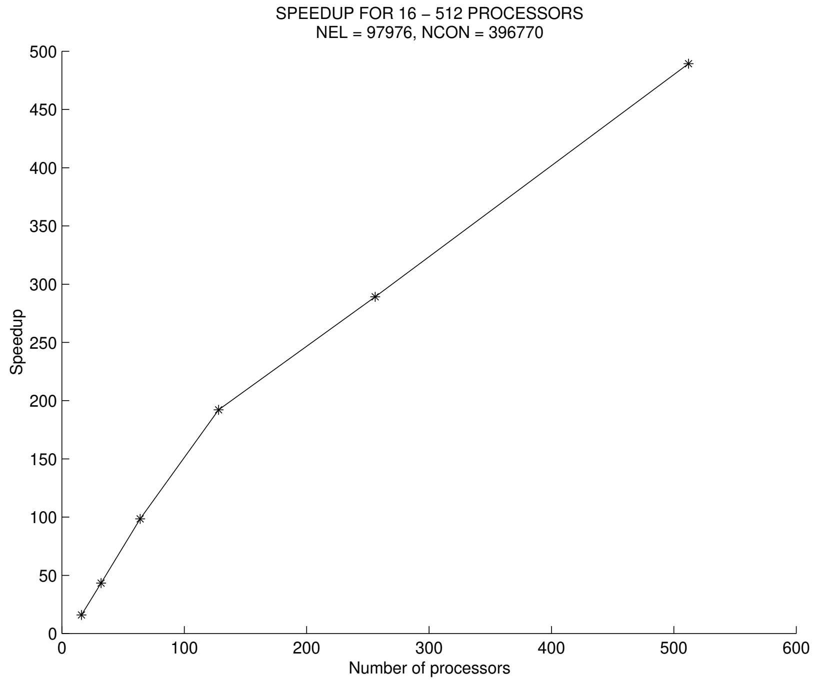


Figure 7: Execution time and parallel speedup on 3D problem for 16, 32, 64, 128, 256 and 512 processors on the Cray T3E-900.

5.2 An Unstructured Problem

In the ideal case, the problem can be evenly divided among the processors not only with approximately the same number of *internal* grid blocks per processor, but also roughly the same number of *external* blocks per processor. Our problems, however, are very unstructured, which means that the partitioning can not be made even in these both aspects.

This leads, for example, to imbalances between the number of external elements per processor when the internal blocks are evenly distributed. For the 3D problem on 512 processors, the average number of external grid blocks is 234 but the maximum number of external blocks for any processor is 374. It follows that at least one processor will have 60% higher communication volume than the average processor (assuming the communication volume to be proportional to the number of external blocks). Note here that the average number of internal grid blocks is 191 for the same case. This means that the average processor actually has more external blocks that internal blocks. Finally, the average number of neighboring processors is 12.59 and the maximum number of neighbors for any processor is 25.

Altogether this indicates that the communication pattern is irregular and that the amount of communication is becoming significant both in terms of number of messages and total communication volume. At the same time, the amount of computations that can be performed without external elements is becoming fairly small.

Despite of these difficulties, the parallel implementation show ability to efficiently use a large number of processors: every half second wall clock time, on 512 processors, a new linear system of size 293,928 $\times$ 293,928 with 8,023,644 non-zero elements is generated and solved. This includes the time for the numerical differentiation for all elements of the Jacobian matrix, the 3×3 Block Jacobi scaling for each block row, the ILUT factorization for the domain decomposition based preconditioner, and a number of BICGSTAB iterations.

5.3 Analysis of Work Load Variations

Several issues need to be considered when analyzing the performance as the number of processors is increased. First, the sizes of the individual tasks to be performed by the different processors is decreased, giving an increased communication to computation ratio, and the relative load imbalance is also likely to increase. In addition, we may find variations in how the time discretization is performed (the number of time steps) and the number of iterations in the Newton process and the linear solver. In order to conduct a more detailed study, we present a summary of iteration counts and timings for the two test problems in Table 2.

The table shows the average number of Newton iterations per time step, and the average number of iterations in the linear solver per time step and per Newton step, as well as the total number of time steps, Newton iterations, and iterations in the linear solver. We recall that the linear system solve is the most time consuming operation and the computation of the Jacobian matrix is the second largest time consumer. Both of these operations are performed once for each Newton step.

Table 2: Iteration counts and execution times for the 2D and 3D test problems.

2D PROBLEM

	2	4	8	16	32	64	128	256
Partitioning algorithm	*VK-way*	*VK-way*	*K-way*	*Rec.*	*Rec.*	*Rec.*	*K-way*	*Rec.*
#Time steps	104	120	104	104	104	94	94	103
Total #Newton iterations	645	869	669	653	663	697	620	637
#Newton iterations/Time step	6.20	7.24	6.43	6.28	6.38	7.41	6.60	6.18
Total #Lin. solv. iterations	8640	16528	10934	9888	11011	11282	11894	19585
#Lin. solv. iter/Newton step	13.40	19.02	16.34	15.14	16.61	18.46	19.18	30.75
#Lin. solv. iter/Time step	83.1	137.1	105.1	95.1	105.9	120.0	126.5	190.1
Time spent on Lin. solv. (secs)	5170	3201	1040	460	242	129	85	104
Time spent on other (secs)	2334	1550	629	303	157	75	41	23
Total time (secs)	7504	4750	1669	762	399	204	126	127

3D PROBLEM

	16	32	64	128	256	512
Partitioning algorithm	*K-way*	*Rec.*	*K-way*	*K-way*	*Rec.*	*Rec.*
#Time steps	154	149	143	137	185	166
Total #Newton iterations	632	606	585	561	708	646
#Newton iterations/Time step	4.10	4.07	4.09	4.09	3.83	3.89
Total #Lin. solv. iterations	8720	10275	9357	10362	14244	14487
#Lin. solv. iter/Newton step	13.80	16.96	15.99	18.47	20.12	22.43
#Lin. solv. iter/Time step	56.6	69.0	65.4	75.6	77.0	87.3
Time spent in Lin. solv. (secs)	7470	2464	995	510	344	224
Time spent on other (secs)	2585	1255	639	327	212	105
Total execution time (secs)	10055	3718	1634	837	556	329

For both problems we note that some variations occur in the time discretization when the problem is solved on different number of processors. Similar behavior has been observed, for example, when using different linear solvers in the serial version of TOUGH2. The variations in time discretization leads to variations both in the number of time steps needed and the number of Newton iterations required. Notably, the 4 processors execution on the 2D problem requires 15% more time steps, 35% more Newton steps, and 91% more iterations in the linear solver compared to the execution on 2 processors. This increase of work fully explains the low speedup on 4 processors. Similar variations in the amount of work also contribute to a very good speedup for some cases.

However, the figures in Table 2 alone do not fully explain the super-linear speedup observed for some cases. We will therefore continue our study by looking at the performance of the linear solver. Before doing that, however, we show some examples that motivates this continued study, i.e., cases where the speedup actually is higher than we would expect from looking at iteration counts only.

For example, on the 2D problem the speedup on 8 processors is 12.4% larger than maximum expected (i.e., 8.99 vs. 8.00), but compared to the execution on two processors, the 8 processor execution actually requires slightly more Newton iterations and iterations in the linear solver. The number of time steps is the same for both tests. When doubling the number of processors from 8 to 16, we see another factor of 2.19 in speedup, even though the reduction in number of Newton iterations and iterations in the linear solver is only 2.4% and 9.6%, respectively. The speedup on the 3D problem from 16 to 32 processors (2.70) and from 32 to 64 processors (2.28) is also higher than what would be expected by looking at Table 2 alone.

So far, we can summarize the following observations for the two problems. The unstructured nature of the problem naturally leads to variations in the work load between different tests. This alone explains some of the speedup anomalies observed, but for a couple of cases, it is evident that are other issues to be investigated. We therefore continue this study by focusing on the performance of the linear solver and the preconditioner.

5.4 Performance of Preconditioner and Linear Solver

A breakup of the speedup in one part for the linear solver (including preconditioner) and one for all other computations (mainly assembly of the Jacobian matrix) is presented for both problems in Figure 8. The figure illustrates that the super-linear speedup for the whole problem follows from super-linear speedup of the linear solver. Note that the results presented are for the total time spent on these parts, i.e., a different number of linear systems to be solved or a difference in the number of iterations required to solve a linear system affect these numbers.

The speedup of the "other parts" is close to p for all tests on both problems, and this is also an indication that this part of the computation may show good performance also for larger number of processors. The slight decrease on 256 and 512 processors for the 3D

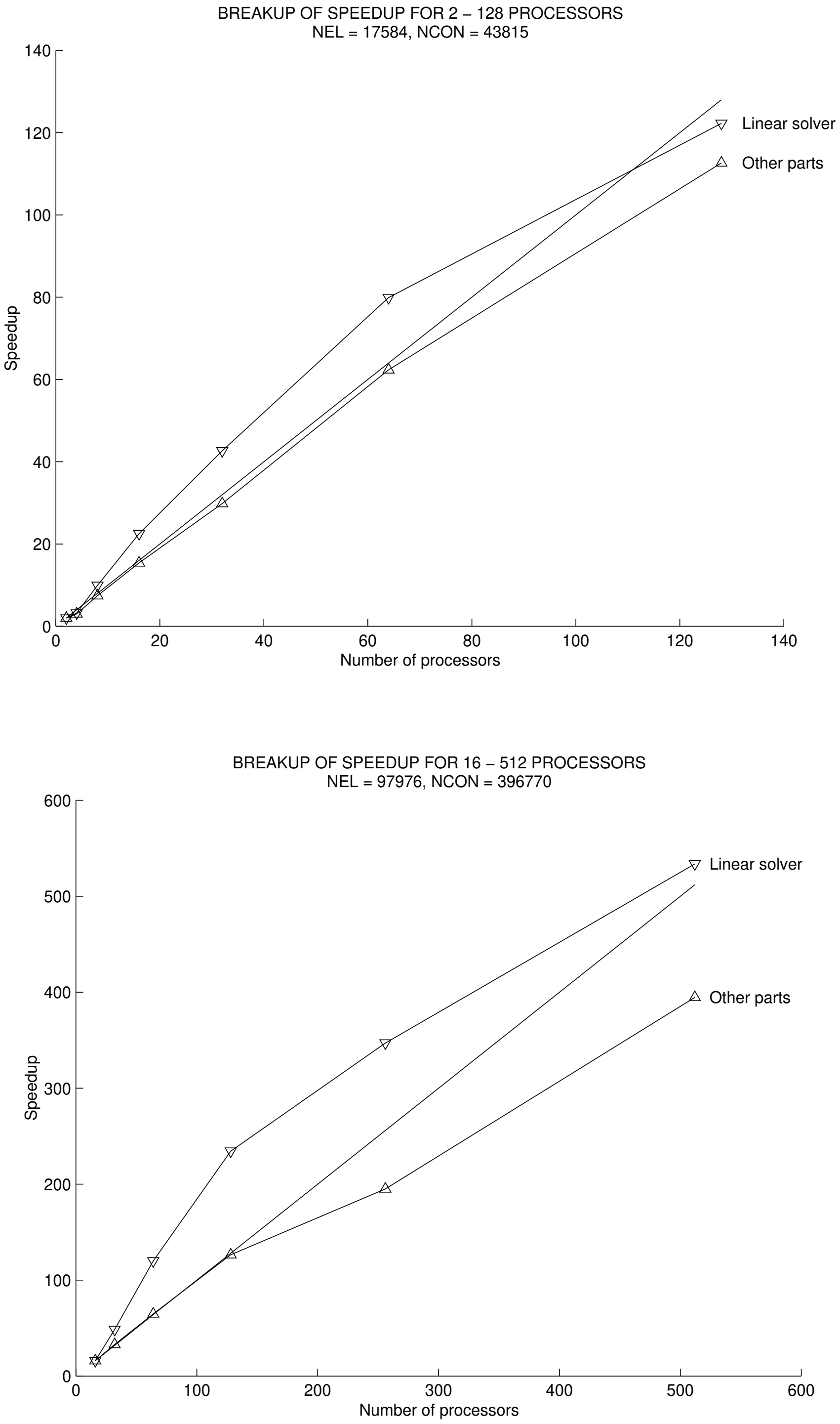


Figure 8: Breakup of speedup for the 2D and 3D problems in one part for the linear solver (marked "∇") and one part for all other computations (marked "△"). The ideal speedup is defined by the straight line.

problem is due to increased number of time steps.

We conclude that the performance of "the other parts" is satisfying and that it needs no further explanations. We continue with the study of the super-linear speedup of the linear solver.

5.4.1 Effectiveness of the Preconditioner

The preconditioner is crucial to the number of iterations per linear system solved. The domain decomposition based process is expected to become less efficient as the number of processors increases. The best effect of the preconditioner is expected when the whole matrix is used in the factorization, but in order to achieve good parallel performance, the size for the preconditioning operation on each processor is restricted to its local subdomain. On average, the matrix used in the preconditioning by each processor is $\frac{n}{p} \times \frac{n}{p}$, where n is the size of the whole (global) matrix and p is the number of processors. The reduced effectiveness follows naturally from the smaller subdomains, i.e., the decreased size of the matrices used in the preconditioner, since only diagonal blocks are used to calculate an approximate solution.

The number of iterations required per linear system for the two test problems confirms this theory (see Table 2). For both test problems the number of iterations required per linear system increases with the number of processors (with exceptions for going from 4 to 8 and 8 to 16 processors for the 2D problem and 32 to 64 for the 3D problem).

We have also included the results for 256 processors on the 2D problem in Table 2. We note an increase in the number of iterations per linear system by more than 50% compared to 128 processors. It is clear that the preconditioner does not do a very good job when the number of processors is increased to 256. By introducing one level of overlapping (Additive Schwarz) in the domain decomposition based preconditioner on 256 processors, the number of iterations in the linear solver is reduced to the same order as for smaller number of processors. This is however done at the additional cost for performing the overlapping and the overall time is roughly unchanged.

Our main observation is that despite the overall increasing number of iterations in the linear solver for increasing number of processors, the speedup of the linear solver is higher than what would normally be expected up to a certain number of processors. The increased number of iterations per linear system for larger number of processors is obviously following from the reduced effectiveness of the preconditioner. In the following sections, we will conclude the performance analysis by investigating the parallel performance of the actual computations performed during the preconditioning and linear iteration processes.

5.4.2 Performance of the Preconditioner

Another effect of the decreased sizes of the subdomains in the domain decomposition based preconditioner is that the total amount of work to perform the incomplete LU factorizations becomes significantly smaller as the number of processors is increased.

For example, as the number of processor is doubled, the size of each processor's local matrix in the ILUT factorization is decreased by a factor of 4, on average from $\frac{n}{p} \times \frac{n}{p}$ to $\frac{n}{2p} \times \frac{n}{2p}$. Hence will the amount of work per processor be reduced a factor between 2 and 8 depending on the sparsity structure. Hence, the amount of work per processor is reduced faster than we normally expect when we assume the ideal speedup to be 2.

Figure 9 gives a further breakup of the speedup, now with the speedup for the linear solver separated into one part for the preconditioner (i.e., the ILUT factorization) and one for the other parts of the linear solver.

The preconditioner shows a dramatic improvement of the performance as the number of processors increases, following naturally from the decreased work in the ILUT factorization. As we continue, this will turn out to be the single most important explanation for the sometimes super-linear speedup.

5.4.3 Performance of the Linear Iterations

We have explained the super-linear speedup of the preconditioning part of the linear solver, presented in Figure 9, but we also observe only a modest speedup of the other parts of the computation.

The low speedup for the other parts is partly explained by the increased number of iterations, as seen in the previous section. Increased communication to computation ratio and slightly increased relative load imbalance are other factors.

Already the figures presented in Section 5.2 showing large number of external elements per processor and an imbalance in the number of external elements per processor presented for the 3D problem on 512 processors indicated that the communication to computation ratio would eventually become large. The performance obtained for the iteration process in the linear solver supports this observation.

5.4.4 Impacts on the Overall Performance

In order to fully understand the total (combined) effect of the super-linear behavior of the preconditioner and the moderate speedup of the other parts of the linear solver, we now only need to investigate how large proportion is spent on preconditioning out of the total time required for solving the linear systems. This is illustrated in Table 3.

As the number of processors becomes large, the amount of time spent on preconditioning becomes small compared to the time spent on iterations, whereas the relation is the opposite for 2 processors on the 2D problem and for 16 processors on the 3D problem. For example, for 16 processors on the 3D problem 88.4% of the time in the linear solver was spent on the factorization for the preconditioner, whereas the corresponding number is only 39.4% for 512 processors. As long as the preconditioner consumes a large portion of the time, its super-linear speedup will have significant effects on the overall performance of the implementation.

It is evident, that up to a certain number of processors, the super-linear speedup of the

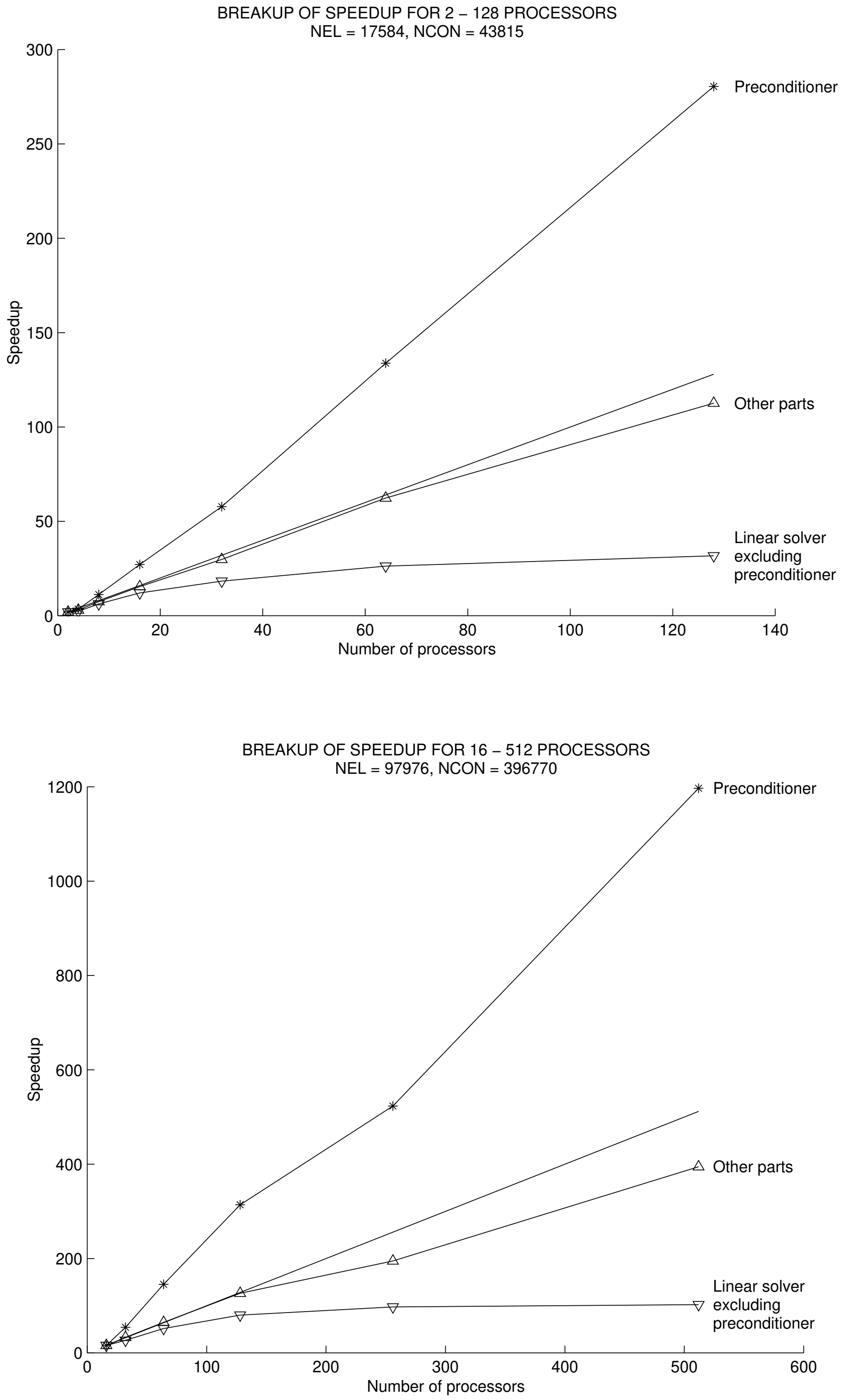


Figure 9: Breakup of speedup for the 2D and 3D problems in one part for the linear solver excluding the ILUT factorization for the preconditioner (marked "∇"), one part for ILUT factorization (marked "*") and one part for all other computations (marked "$\triangle$"). The ideal speedup is defined by the straight line.

Table 3: Time spent on preconditioning as percentage of total time spent in linear solver.

2D PROBLEM

#Processors	2	4	8	16	32	64	128
Percentage	83.5%	77.1%	73.9%	69.2%	61.6%	49.9%	36.4%

3D PROBLEM

#Processors	16	32	64	128	256	512
Percentage	88.4%	78.9%	73.0%	66.0%	58.6%	39.4%

incomplete LU factorization in the domain decomposition based preconditioner is sufficient to give super-linear speedup for the whole application. As the number of processor becomes large, the factorization consumes a smaller proportion of the execution time, and hence, it's super-linear behavior has less impact on the overall performance. Instead, there are other issues that become more critical for large number of processors, such as the increased number of iterations in the linear solver.

6 Conclusions

This contribution presents a parallel prototype implementation that shows to efficiently use up to at least 512 processors of the Cray-T3E system. The implementation is made with flexibility to use different linear solvers, preconditioners, and grid partitioning algorithms, as well as alternative Eos modules for solving different problems. Computational experiments on real application problems show high speedup for up to 128 processors on a 2D problem and up to 512 processors on a 3D problem.

The results are accompanied by an analysis that explains the good parallel performance observed. It also explains some minor variations in performance following from the unstructured nature of the problem and some super-linear speedups following from decreased work in the preconditioning process.

The results also illustrate the trade-off between the time spent on preconditioning and the effect of its result. With the objective to minimize the wall clock execution time, we note that, for these particular problems, smaller subdomains could be used, at least on small number of processors.

We have seen some variations in performance in tests using three different partitioning algorithms. Some of these variations clearly follow from variations in the amount of work required, e.g., due to differences in the time discretization. Further analysis is required in order to determine whether these variations follow some particular pattern or if they are only a result from unpredictable circumstances.

The problems we are targeting in the near future are larger both in terms of number of blocks and number of equations per block. Moreover should the simulation time be significantly longer. With increased problem size we expect to be able to efficiently use an even larger number of processors (if available), and longer simulations should not directly affect the parallel performance.

Future investigations include studies of alternative non-linear solvers and further studies of the interplay between the time stepping procedure, the non-linear systems, and the linear systems. Evaluations of different linear solvers, preconditioners, parameter settings, and partitioning algorithms would be of general interest and may help to further improve the performance of this particular implementation.

7 Acknowledgements

We thank Karsten Pruess, the author of the original TOUGH2 software, for valuable discussions during this work.

This work is supported by the Director, Office of Science, Office of Laboratory Policy and Infrastructure, of the U.S. Department of Energy under contract number DE-AC03-76SF00098. This research uses resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

References

[1] G.S. Bodvarsson, T.M. Bandurraga, and Y.S. Wu. The site-scale unsaturated zone model of Yucca Mountain, Nevada, for the viability assessment. Yucca Mountain Characterization Project Report LBNL-40376, UC-814, Earth Sciences Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 1997.

[2] E. Elmroth, C. Ding, Y.-S. Wu, and K. Pruess. A parallel implementation of the TOUGH2 software package for large scale multiphase fluid and heat flow simulations. In *Proceedings of Supercomputing '99*. ACM, 1999.

[3] Message Passing Forum. A Message-Passing Interface standard. *International Journal of Supercomputing Applications and High Performance Computing*, 8(3–4), 1994.

[4] S.A. Hutchinson, L.V. Prevost, J.N. Shadid, C. Tong, and R.S. Tuminaro. Aztec user's guide. Version 2.0. Technical report, Massively Parallel Computing Research Center, Sandia National Laboratories, Albuquerque, NM, 1998.

[5] G. Karypsis and V. Kumar. METIS. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4.0. Technical report, Department of Computer Science, University of Minnesota, 1998.

[6] K. Pruess. TOUGH user's guide. Technical Report LBNL-29400, UC-251, Earth Sciences Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 1987.

[7] K Pruess. TOUGH2—a general-purpose numerical simulator for multiphase fluid and heat flow. Technical Report LBNL-29400, UC-251, Earth Sciences Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 1991.

[8] K. Pruess (editor). *Proceedings of the TOUGH Workshop '98.* Technical Report LBNL-41995, Conf-980559, Earth Sciences Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 1998.

[9] Y. Saad. ILUT: A dual threshold incomplete ILU preconditioner. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[10] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[11] H. Van der Vorst. BICGSTAB: A fast and smoothly converging variant of the BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. and Stat. Comp.*, 13:631–644, 1992.