SPE 66343

Parallel Computing Techniques for Large-Scale Reservoir Simulation of Multi-Component and Multiphase Fluid Flow

K. Zhang, Y. S. Wu, SPE, C. Ding, K. Pruess, SPE, and E. Elmroth, Lawrence Berkeley National Laboratory

This paper was prepared for presentation at the SPE Reservoir Simulation Symposium held in Houston, Texas, 11–14 February 2001.

Abstract

Massively parallel computing techniques can overcome limitations of problem size and space resolution for reservoir simulation on single-processor machine. This paper reports on our work to parallelize a widely used numerical simulator, known as TOUGH2, for nonisothermal flows of multi-component, multiphase fluids in three-dimensional porous and fractured media. We have implemented the TOUGH2 package on a Cray T3E-900, a distributed-memory massively parallel computer with 695 processors. For the simulation of large-scale multicomponent, multiphase fluid flow, the requirements for computer memory and computing time are extensive. Because of the limitation of computer memory in each PE (processing element), we distribute not only computing time but also the memory requirement to different PEs. In this study, the METIS software package for partitioning unstructured graph and meshes is adopted for domain partitioning, and the Aztec linear solver package is used for solving linear equation systems. The efficiency of the code is investigated through the modeling of a three-dimensional variably saturated flow problem, which involves more than one million gridblocks. The execution time and speedup are evaluated through comparing the performance of different numbers of processors. The results indicate that the parallel code can significantly improve capacity and efficiency for large-scale simulations.

Introduction

TOUGH2[1, 2] is a general-purpose numerical simulation program for multi-dimensional, multiphase, multicomponent heat and fluid flows in porous and fractured media. The code is written in standard ANSI FORTRAN 77. Since its release in 1991, the program has been used worldwide in geothermal reservoir engineering, nuclear waste isolation, environmental assessment and remediation, and modeling flow and transport in variably saturated media. The numerical scheme of the TOUGH2 code is based on the integral finite difference (IFD) method. The conservation equations involving mass of air, water, chemical components and thermal energy are discretized in space using the IFD method. Time is discretized fully implicitly using a first-order backward finite difference scheme. The discretized nonlinear system of finite difference equations for mass and energy balances are solved simultaneously using the Newton/Raphson iterative scheme. For the basic version (i.e., single CPU), the code is equipped with both direct and iterative solvers.[3]

The development of parallel computers has made it possible to conduct large-scale reservoir simulations. In the past decade, the total number of gridblocks used in a typical reservoir simulation increased from thousands to millions.[4] One of the most popular parallel computer architectures is the distributed-memory machine, the massively parallel processor (MPP) computer, which can be made up of hundreds to thousands of processors. Elmroth et al.[5] developed a parallel prototype scheme for the TOUGH2 code and implemented the computing time distribution on MPP computer. Their investigation indicates that a parallel code can dramatically enhance computational efficiency.

The present work presents the further progress in reducing memory requirement and improving computation efficiency, including the optimization for solving extremely large reservoir simulation problems. The parallelization of the TOUGH2 code was implemented on a Cray T3E-900, an MPP computer. The parallel code was developed from the original TOUGH2 code by introducing the message-passing interface (MPI) library.[6] MPI is a standard procedure for message passing that allows data transfer from one processor to another. The parallel implementation first partitions an unstructured simulation domain using the METIS graph partitioning programs.[7] The spatially discretized nonlinear equations describing the flow system are then set up for each partitioned part at each time step. These equations are solved with the Newton iteration method. In each Newton step, a nonsymmetric linear equation system is formed for each part of the domain and is then solved using a preconditioned

iterative solver selected from the Aztec linear solver package.[8] During each Newton iteration, the linearized equation systems must be updated with the updating in primary variables. Updating the left-hand side Jacobian matrix requires communication between different processors for data exchange across the partitioning borders. By distributing the computation time and memory requirement to processors, the parallel TOUGH2 code allows more accurate representation of reservoirs because of its ability to include more detailed information on a refined grid system.

The significant enhancement on computational efficiency in the parallel TOUGH2 code is demonstrated through modeling of a field flow problem. The code has been used to develop a three-dimensional (3-D) model of multiphase fluid flow in variably saturated fractured rocks. The 3-D model uses more than 10^6 gridblocks and 4×10^6 connections (interfaces) to represent the unsaturated zone of the highly heterogeneous, fractured tuffs of Yucca Mountain, Nevada, a potential underground repository for high-level radioactive wastes. Numerical simulation of the unsaturated zone flow system at Yucca Mountain has become a standard tool in site-characterization investigation.[9] However, the 3-D, site-scale unsaturated flow models, developed since the early 1990s,[10] in general use very coarse numerical grids primarily because of limitation in computational capacity.

In this paper, we discuss the main issues addressed in the implementing TOUGH2 on the massively parallel T3E-900 machine. We then present an example problem for unsaturated flow at the Yucca Mountain site, Nevada, which has more than 1 million grid blocks. This problem is used to evaluate speedup from code parallelization, and to confirm the solution accuracy of the massively parallel code by comparison with results from a single-processor machine.

Parallel Implementation

As discussed above, the TOUGH2 code using an IFD method[11, 12] solves mass and energy balance equations of fluid and heat flow in a multiphase, multicomponent system. The IFD approach avoids any reference to a global system of coordinates and thus offers the advantages of being applicable to regular or irregular discretization in multiple dimensions. However, the flexibility in IFD formation gridding makes a model grid that intrinsically unstructured, which must be taken into account by a parallelization scheme.

In the basic version of the TOUGH2 code, the discretization in space and time using the IFD leads to a set of strongly coupled nonlinear algebraic equations, which is linearized by the Newton method. Within each Newton iteration, the Jacobian matrix is first calculated by numerical differentiation, the resulting system of linear equations then solved using an iterative linear solver with preconditioning. Time steps can be automatically adjusted (increased or decreased) during a simulation run, depending on the convergence rate of the iteration process. For a TOUGH2 simulation, the most time-consuming steps of the execution consist of two parts: (1) solving the linear system of equations and (2) assembling the Jacobian matrix. Consequently, one of the most important aims of the parallel TOUGH2 code is to distribute computing time for these two parts. The main schemes implemented in the parallel code include grid partitioning, grid reordering, optimizing data input, assembly of the Jacobian matrix, and solving the linear system. The first stage of the work was summarized by Elmroth et al.[5] The following sections give an overview of the most important parallel implementation procedures.

Grid Partitioning and Gridblock Reordering. Efficient and effective methods for partitioning unstructured grid domains are critical for successful parallel computing schemes. Large-scale numerical simulations on parallel computers require the distribution of gridblocks to different processing elements. This distribution must be carried out such that the number of gridblocks assigned to each PE is the same and the number of adjacent blocks duplicated and copied to each PEs is minimized. The goal of the first condition is to balance the computation efforts among the PEs; the goal of the second condition is to minimize the time-consuming communication resulting from the placement of adjacent blocks to different processors.

In a TOUGH2 simulation, a model domain is represented by a set of gridblocks (elements), and the interfaces between every two gridblocks are represented by connections. The entire connection system of gridblocks is defined through input data. From the connection information, an adjacency matrix can be constructed. The adjacency structure of the model meshes is stored using a compressed storage format (CSR). In this format, the adjacency structure of a domain with n gridblocks and m connections is represented using two arrays, *xadj* and *adj*. The *xadj* array has a size of $n+1$ whereas the *adj* array has a size of $2m$.

The adjacency structure of the model grids is stored in a compressed format which can be described as follows. Assuming that element numbering starts from *1*, then the adjacency list of element *i* is stored in an array *adj*, starting at index *xadj(i)* and ending at index *xadj(i+1)-1*. That is, for each element *i*, its adjacency list is stored in consecutive locations in the array *adj*, and the array *xadj* is used to point to where it begins and where it ends. Figure 1a shows the connection of a 12-elements domain and Figure 1b illustrates its corresponding CSR format arrays.

We use three partitioning algorithms implemented in the METIS package version 4.0[7]. The three algorithms are here denoted the *K-way*, the *VK-way*, and the *Recursive* partitioning algorithm. *K-way* is used for partitioning a graph into a large number of partitions (greater than 8). The objective of this algorithm is to minimize the number of edges that straddle different partitions. If a small number of partitions are desired, the *Recursive* partitioning method, a recursive bisection algorithm, should be used. *VK-way* is a modification of *K-way* and its objective is to minimize the total communication volume. Both *K-way* and *VK-way* are multilevel partitioning algorithms.

Figure 1a shows a scheme of partitioning a sample domain into three parts. Gridblocks are assigned to particular processors through partitioning methods and reordered by each processor to a local ordering. Elements corresponding to these blocks are explicitly stored on the processor and are defined by a set of indices referred to as the processor's *update* set. The *update* set is further divided into two subsets: *internal* and *border*. Vector elements of the *internal* set are updated using only information on the current processor. The *border* set consists of blocks with at least one edge to a block assigned to another processor. The *border* set includes blocks that would require values from other processors to be updated. The set of blocks that are not in the current processor, but needed to update components in the *border* set, is referred to as an *external* set. Table 1 shows the partitioning results and one of the local numbering schemes for the sample problem presented in Figure 1a.

The local numbering of gridblocks is done to facilitate the communication between processors. The numbering sequence is *internal* blocks followed by *border* blocks and finally by the *external* set. In addition, all *external* blocks from the same processor are in consecutive order.

Similar to vectors, a subset of matrix with non-zero entries is stored on each processor, In particular, each processor stores only those rows, that correspond to its *update* set. These rows form a submatrix whose entries correspond to variables of both the *update* set and the *external* set defined on this processor.

Input Data Organization. The input data for reservoir simulations include hydrogeologic parameters and constitutive relations of porous media, such as absolute and relative permeability, porosity, capillary pressure, thermophysical properties of fluid and rock, as well as initial and boundary conditions of the system. In addition, a numerical code requires specification of space-discretized geometric information (grid) and various program options such as computational parameters and time-stepping information. For a typical, large-scale, three-dimensional model, computer memory of several gigabytes is generally required. Therefore, the need arises to distribute the memory requirement to all processors.

Each processor has a limited space of memory available. To make efficient use of the memory of each processor, the input data files of the TOUGH2 code are organized in sequential format. There are two groups of large data blocks within a TOUGH2 mesh file: one with dimensions equal to the number of grid blocks, the other with dimensions equal to the number of connections (interfaces). Large data blocks are read one by one through a temporary full-size array and then distributed to PEs one by one. This method avoids storing all input data in a single processor and greatly enhances the I/O efficiency. The I/O efficiency is further improved by storing the input data in binary files. The data input procedures can be schematically outlined as follows:

In PE0:

Open a data file
Read first parameter for all blocks (total NEL blocks) into array Temp(NEL)
Do i=1,TotalPEs
Call MPI_SEND(…) to send the appropriate part of Temp(NEL) to PEi.
End do
Read second parameter for all blocks into array Temp(NEL)
Do i=1,TotalPEs
Call MPI_SEND(…) to send the appropriate part of Temp(NEL) to PEi.
End do
……
Repeat for all parameters that need to be read from data file for all gridblocks.
Read first parameter for all connections (NCON) into array Temp(NCON)
Do i=1,TotalPEs
Call MPI_SEND(…) to send the appropriate part of Temp(NCON) to PEi.
End do
Read second parameter all connections into Temp(NCON)
Do i=1,TotalPEs
Call MPI_SEND(…) to send the appropriate part of Temp(NCON) to PEi.
End do.
……
Repeat for all parameters that need to be read from data file for all connections.
Close data file.

In PE1, PE2, ……, PEn:
Allocate required memory space for current PE .
Call MPI_RECV(…) to receive the part of data that belongs to current PE from PE0.

Certain parts of the parallel code require full-connection information, such as for domain partitioning and local-connection index searching. These parts can be the bottleneck of memory requirement for solving a large problem. Since the full-connection information is used only once at the beginning of a simulation, it may be better handled in a preprocessing procedure.

Assembly and Solution of Linear Equation Systems. The discrete mass and energy balance equations solved by the TOUGH2 code can be written in residual form:[1,2]

$$R_n^\kappa(x^{t+1}) = M_n^\kappa(x^{t+1}) - M_n^\kappa(x^t) - \frac{\Delta t}{V_n}\{\sum_m A_{nm} F_{nm}^\kappa(x^{t+1}) + V_n q_n^{\kappa,t+1}\} \quad \text{……..……..……(1)}$$

where the vector x^t consists of primary variables at time t, R_n^κ

is the residual of component κ for block n, M denotes mass per unit volume for a component, V_n is the volume of the block n, q denotes sinks and sources, Δt denotes current time step size, $t+1$ denotes the current time, A_{nm} is the interface area between blocks n and m, and F_{nm} is the flow between them. Equation (1) is solved using Newton-Raphson iteration method, leading to

$$-\sum_i \left.\frac{\partial R_n^{\kappa,t+1}}{\partial x_i}\right|_p (x_{i,p+1} - x_{i,p}) = R_n^{\kappa,t+1}(x_{i,p}) \quad \ldots\ldots(2)$$

where $x_{i,p}$ represents the value of ith primary variable at pth iteration step. The Jacobian matrix as well as the right-hand side of (2) needs to be recalculated at each Newton iteration. The computational efforts are extensive for a large simulation problem. In the parallel code, the assembly of linear equation system (2) is shared by all the processors. Each processor is responsible for computing the rows of the Jacobian matrix that correspond to blocks in the processor's *update* set. Computation of the elements in the Jacobian matrix is performed in two parts. The first part consists of computations relating to individual blocks. Such calculations are carried out using the information stored on current processor and communications to other processors are not necessary. The second part includes all computations relating to the connections. The elements in the *border* set need information from the *external* set, which requires communication between neighbor processors. Before performing these computations, an exchange of relevant variables is required. For the elements corresponding to *border* set blocks, one processor sends these elements to different but related processors, which receive these elements as *external* blocks.

The Jacobian matrix for local gridblocks in each processor is stored in the distributed variable block row (DVBR) format,[8] a generalization of the VBR format. All matrix blocks are stored row-wise, with the diagonal blocks stored first in each block row. Scalar elements of each matrix block are stored in column major order. The data structure consists of a real vector and five integer vectors, forming the Jacobian matrix. The detail explanation for the DVBR data format can be found from reference[8].

The final, local linear equation systems are solved by using the Aztec linear solver package[8]. We can select different solvers and preconditioners from the package. The available solvers include conjugate gradient, restarted generalized minimal residual, conjugate gradient squared, transposed-free quasi-minimal residual, and bi-conjugate gradient with stabilization methods. The results presented in this paper have been obtained using the stabilized bi–conjugate gradient method with block Jacobian scaling and a domain decomposition preconditioner (additive Schwarz). In block Jacobian scaling, the block size corresponds to the VBR blocks, which are determined by the equation number of each gridblock. Detailed discussions on preconditioning and scaling scheme were presented by Elmroth et al.[5]

During a simulation, the time steps are automatically adjusted (increased or reduced), depending on the convergence rate of the iteration process in the current step. In the parallel version code, the time-step size found in the first processor (master processor, named PE0) is applied to all processors. The convergence rates may be different in different processors. Only when all processors reach stopping criteria will the time march to the next step.

Final solutions are derived from all processors and transferred to master processor for output. Results for the connections that cross the boundary of two different processors are obtained by averaging the solutions from the two processors.

Data Exchange Between Processors. Data communication between processors is an essential component of the parallel TOUGH2 code. Although each processor solves the linearized equations of the local blocks independently, communication between neighboring processors is necessary to update and solve the entire equation system. The data exchange between processors is implemented through the EXCHEXTERNAL subroutine. When this subroutine is called by all processors, an exchange of vector elements corresponding to the *external* set of the gridblocks will be performed. During time stepping or a Newton iteration, an exchange of external variables is also required for the vectors containing the secondary variables and the primary variables. Detailed discussion of the implementation of data exchange can be found in Elmroth et al.[5]

Program Structure. The parallel version of TOUGH2 has almost the same program structure as the original version of the software, but solves a problem using multiple processors. We introduce dynamic memory management, modules, array operations, matrix manipulation, and other FORTRAN 90 features to the parallel code. MPI is used for message passing. Another important modification to the original serial code is in the subroutine of time-step looping. This subroutine provides the general control of problem initialization, grid partitioning, data distribution, memory-requirement balancing among all processors, time stepping, and output. All data input and output are carried out through the master processor . The most time-consuming efforts, such as assembling the Jacobian matrix, updating thermophysical parameters, and solving the linear equation systems, are distributed to all processors. The memory requirements are also distributed to all processors. Distribution of computing time and memory requirements is essential for achieving a capacity for solving large-scale field problems. Figure 2 gives an abbreviated overview of the program flow chart.

Application On Yucca Mountain Problem

Performance of the parallel code was evaluated and demonstrated through a three-dimensional flow simulation of the unsaturated zone at Yucca Mountain, Nevada. The

problem is based on the site-scale model developed for investigations of the unsaturated zone at Yucca Mountain, Nevada.[9,10] It concerns unsaturated flow through fractured rock using a 3-D, unstructured grid and a dual permeability conceptualization for handling fracture-matrix interactions. The unsaturated zone of Yucca Mountain is being investigated as a potential subsurface repository for storage of high-level radioactive wastes. The model domain of the unsaturated zone encompasses approximately 40 km^2 of the Yucca Mountain area, is between 500 and 700 m thick, and overlies a relatively flat water table.

The 3-D model domain as well as a 3-D irregular numerical grid used for this example is shown for a plan view in Figure 3. The model grid uses relatively refined gridding in the middle, repository area, and includes several nearly vertical faults. The grid has about 9,800 blocks per layer for fracture and matrix continua, respectively, and about 60 computational grid layers in the vertical direction, resulting in a total of 1,075,522 gridblocks and 4,047,209 connections. A distributed-memory Cray T3E-900 computer equipped with 695 processors has been used for the simulation. Each processor has about 244 MB available memory and is capable of performing 900 million floating operations per second (MFLOPS).

The ground surface is taken as the top model boundary, and the water table is regarded as the bottom boundary. Both top and bottom boundaries of the model are assumed Dirichlet-type conditions. In addition, on the top boundary, a spatially varying infiltration is applied to describe the net water recharge, with an average infiltration rate of 4.6 mm/yr over the model domain.[10] The properties used for rock matrix and fractures for the dual permeability model, including two-phase flow parameters of fractures and matrix, were estimated based on field tests and model calibration efforts, as summarized in Wu et al. [9]

The linear equation system arising from the Newton iteration of the Yucca Mountain problem is solved by the stabilized bi-conjugate gradient method. A domain decomposition-based preconditioner with ILUT incomplete LU factorization has been selected for preconditioning, and the *K-way* partitioning algorithm has been selected for partitioning the problem domain. The stopping criteria used for the iterative linear solver is

$$\frac{\|r\|_2}{\|b\|_2} \le 10^{-4} \qquad \text{.....(3)}$$

where $\|\cdot\|_2 = \sqrt{(1/n)\sum_{i=1}^{n} r_i^2}$, n is the total number of unknowns, and r and b are the residual and right-hand side, respectively.

Two types of tests were run (1) to examine the accuracy of the parallel code, and (2) to evaluate the code performance and parallelization gains for different numbers of processors. The first test simulates the flow system to steady state. The simulation results for steady state flux through the repository and bottom layer are compared to results previously obtained from simulations on a single-processor machine. The second test used different numbers of processors to simulate the unsaturated flow system for 200 time steps.

Steady State Test. The test problem was designed to test the accuracy of solutions. We have verified the modeling results from the parallel code by comparing the solutions for a smaller grid model using a one-dimensional vertical column. The solutions for the smaller problem were obtained using the original, single-CPU version and the parallel version of the TOUGH2 code. The test presented here provides a further verification of the code for large-scale simulations.

The 3-D test problem was run on 64 processors for 3,684 time steps to reach steady state, recognized when the fluxes going into and leaving the flow system are equal (within a narrow difference). Because of the time limitation of the computer batch system, the whole simulation is divided into five stages. Each stage runs about 700 time steps in less than four hours. The length of a total simulation time is about 10^{11} years when steady state is obtained.

The percolation flux through the repository horizon and below is one of the most important factors considered in evaluation of repository performance. Figures 4 and 5 show the flux distributions along the repository horizon and at the bottom of the simulation domain (the water table). The dark color indicates higher values of percolation fluxes. The flux is defined in the figures as total mass flux through both fractures and matrix. Comparison of the simulation results (Figures 4 and 5) against those using coarse-grid models[10] indicates that the refined-grid model produces results with much higher resolution and more accurate flow distributions at the repository level as well as the water table. In particular, the current, refined model predicts more significant lateral flow in the upper part of the unsaturated zone, above the repository horizon, due to using finer vertical grid spacings in these layers. These modeling results will have direct impact on assessing repository performance. Further simulation results will be reported elsewhere.

Performance Test. In the second test, the problem was solved using 32, 64, 128, 256, and 512 processors, respectively. Because of the automatic time-step adjustment, based even on the same convergence rate of the iteration process, the length of simulation times over 200 time steps using different numbers of processors may be different. However, the computational targets are similar, and comparing the performance of different numbers of processors with the same number of time steps is reasonable for evaluating the parallel code.

Table 2 shows the reduction in the total execution time with an increase numbers of processors. The simulation was run on from 32 processors up to 512 processors by consecutively doubling the number of processors. The results clearly indicate that the execution time is significantly reduced, as the number of processors increases. Table 2 also

shows the time required for different computational tasks using different numbers of processors. When less than 128 processors are used, doubling the processor number will reduce the total execution time by more than half. From the table, we can find that the best parallel performance is in solving-linear equation systems. Data input and output of the program are carried out through a single processor, which will limit the performance of the parallel code for those parts.

Figure 6 illustrates the speedup of the parallel code. The speedup is defined based on the performance of 32 processors as $32T_{32}/T_p$, where T_p denotes the total execution time using p processors. The speedups from 32 to 64, 128, 256, and 512 processors increase by factors of 2.63, 2.16, 1.87 and 1.54, respectively. Super-linear speedup appears during the processor number doubling from 32 to 64, and to 128 with a speedup of 2.63 and 2.16. The overall speedup for 512 processors is 523. The super-linear speedup is mostly due to the preconditioner in solving linear equation system where the time requirement is proportional to n^2, with n being the number of gridblocks in each processor.

In contrast, the time requirement for the startup phase (input, partition, distribution, and initialization) in Table 2 increases when the processor number is doubled from 256 to 512 (instead of decreasing). It indicates that a saturation point has reached. This results from the increase of communication overhead when increasing the number of processors, which cancels the time saving by requiring more processors in this range.

The partitioning algorithm can also significantly impact parallel code performance. The ideal case is that the gridblocks can be evenly distributed among the processors with not only approximately the same number of internal gridblocks, but also roughly the same number of external blocks per processor. For unstructured grids, this ideal situation may be difficult to achieve in practice. However, in our problem gridblocks are almost evenly divided among processors. For example, on 128 processors, the average number of internal blocks is 8,402 at each processor, the maximum number is 8,657 and minimum number is 8,156. It is only about 6% different between the maximum and minimum number. A considerable imbalance arises for the external blocks. In this problem, the average number of external blocks is 2,447, while the maximum number is as large as 3,650 and the minimum as small as 918. This large range indicates that the communication volume can be four times higher for one processor than another. The imbalance in communication volume results in a considerable amount of time wasted on waiting for certain processors to complete their jobs during the solving of equation systems.

In general, the memory capacity of a single processor may be too small to solve a problem with more than one million gridblocks. The distribution of memory requirement among all the processors will solve the storage problem of input data. For the Yucca Mountain one-million block problem, the parallel-computing performance is satisfactory for both computation time and memory requirement.

Conclusions

Massive parallel computing technology has been implemented into the TOUGH2 code for application to large-scale reservoir simulations. In the parallel code, both computing efforts and memory requirements are distributed among and shared by all processors of a multi-CPU computer. This parallel computing scheme makes it possible to solve large simulation problems using a parallel processor computer. The METIS graph partitioning program was adopted for the grid partitioning, and the Aztec package was used for solving the linear equation systems.

The parallel TOUGH2 code has been tested on a Cray T3E system with 512 processors. Its performances are evaluated through modeling flow in the unsaturated zone at Yucca Mountain using different numbers of processors with more than a million gridblocks. The total execution time is reduced from 10,101 seconds on 32 processors to 618 seconds on 512 processors for the field-scale variably saturated flow problem. A super-linear speedup of 523 for 512 processors has been reached. Test results indicate that the overall performance of the parallel code shows significant improvement in both efficiency and ability for large-scale reservoir simulations. The major benefits of the code are that it (1) allows accurate representation of reservoirs with sufficient resolution in space, (2) allows adequate description of reservoir heterogeneities, and (3) enhances the speed of simulation.

Acknowledgment

The authors would like to thank Jianchun Liu and Dan Hawkes for their review of this paper. The authors are grateful to Lehua Pan for his help in designing the 3-D grid used for the test problem. This work was supported by the Laboratory Directed Research and Development (LDRD) program of Lawrence Berkeley National Laboratory. The support is provided to Berkeley Lab through the U. S. Department of Energy Contract No. DE-AC03-76SF00098.

References

1. Pruess, K.: "TOUGH2 – A general-purpose numerical simulator for multiphase fluid and heart flow," Lawrence Berkeley Laboratory Report LBL-29400, Berkeley, CA, 1991.
2. Pruess, K. Oldenburg, C., and Moridis, G.: "TOUGH2 User's Guide, V2.0," Lawrence Berkeley National Laboratory, Berkeley, CA, 1999.
3. Moridis, G. and Pruess, K.: "An enhanced package of solvers for the TOUGH2 family of reservoir simulation codes," *Geothermics* (1998) **27**, No.4, 415-444.
4. Dogru, A. H.: "Megacell reservoir simulation," *JPT* (MAY 2000), 54-60.
5. Elmroth, E., Ding, C., and Wu, Y.: "High performance computations for large scale simulations of subsurface multiphase fluid and heat flow," accepted by The Journal of Supercomputing, 1999.
6. Message Passing Formum: "A message-passing interface standard," International Journal of Supercomputing Applications and High performance Computing, **8**(3-4), 1994.
7. Karypsis, G. and Kumar, V.: "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-

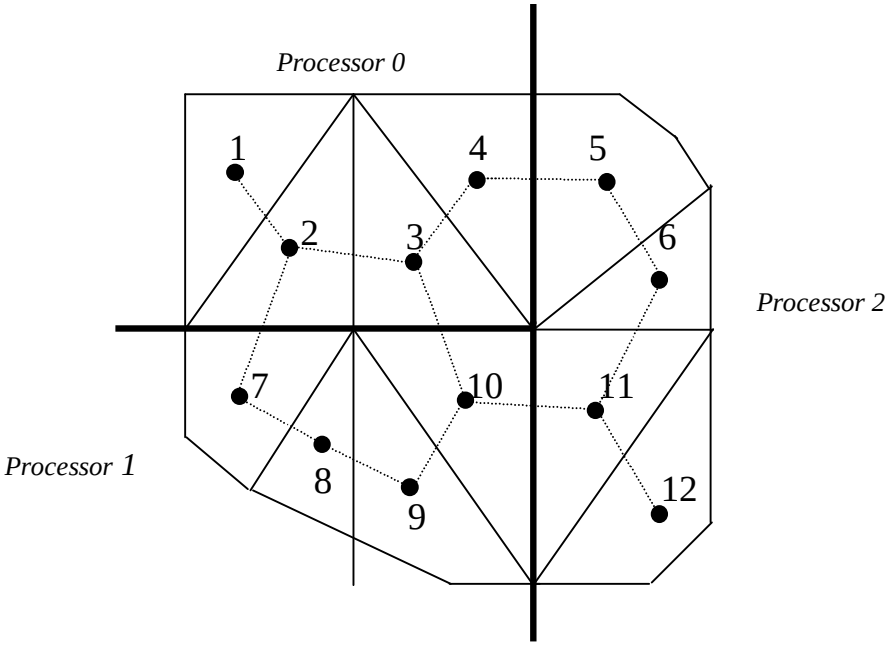
reducing orderings of sparse matrices, V4.0," Technical Report, Department of Computer Science, University of Minnesota, 1998.
8. Tuminaro, R. S., Heroux, M., Hutchinson, S. A., and Shadid J. N.: "Official Aztec user's guide, Ver 2.1," Massively Parallel Computing Research Laboratory, Sandia National Laboratories, Albuquerque, NM, 1999.
9. Wu, Y. S., Liu, J., Xu, T., Haukwa, C., Zhang, W., Liu, H. H., and Ahlers, C. F.: "UZ Flow Models and Submodels," Report MDL-NBS-HS-000006, Berkeley, California: Lawrence Berkeley National Laboratory, Las Vegas, Nevada, CRWMS M&O, 2000
10. Wu, Y.S., Haukwa, C., and Bodvarsson, G. S.: "A Site-Scale Model for Fluid and Heat Flow in the Unsaturated Zone of Yucca Mountain, Nevada," *Journal of Contaminant Hydrology* (1999), **38** (1-3), pp.185-217.
11. Edwards, A. L.: "TRUMP: a computer program for transient and steady state temperature distributions in multidimensional systems," National Technical Information Service, National Bureau of Standards, Spingfield, VA 1972.
12. Narasimhan, T. N. and Witherspoon P. A.: "An integrated finite difference method for analyzing fluid flow in porous media," *Water Resour. Res.* (1976), **12**, 1, 57-64.

Table 1: Example of Domain Partitioning and Local Numbering

		Update		External
		Internal	Border	
Processor 0	Gridblocks	1	2 3 4	5 7 10
	Local numbering	1	2 3 4	5 6 7
Processor 1	Gridblocks	8 9	7 10	2 3 11
	Local Numbering	1 2	3 4	5 6 7
Processor 2	Gridblocks	6 12	5 11	4 10
	Local numbering	1 2	3 4	5 6

Table 2. Breakup of Execution Times (Seconds) for the Yucca Mountain Problem Running 200 Time Steps.

PE number	32	64	128	256	512
Input, partition, distribution, and initialization	592.3	248.1	116.5	84.3	134.3
Update thermophysical parameters, setup Jacobian matrix and save results	2659.2	1420.8	764.6	399.5	260.0
Solve linear equations	6756.7	2078.7	806.6	373.4	188.0
Total execution time	10100.5	3844.3	1780.8	950.6	618.0



(a) A 12-elements domain partitioning on 3 processors

Elements		1	2	3	4	5	6	7	8	9	10	11	12
xadj	1	2	5	8	10	12	14	16	18	20	23	26	27
adj		2	1,3,7	2,4,10	3,5	4,6	5,11	2,8	7,9	8,10	3,9,11	6,10,12	11

(b) CSR format

Figure 1. An example of domain partitioning and CSR format for storing connections

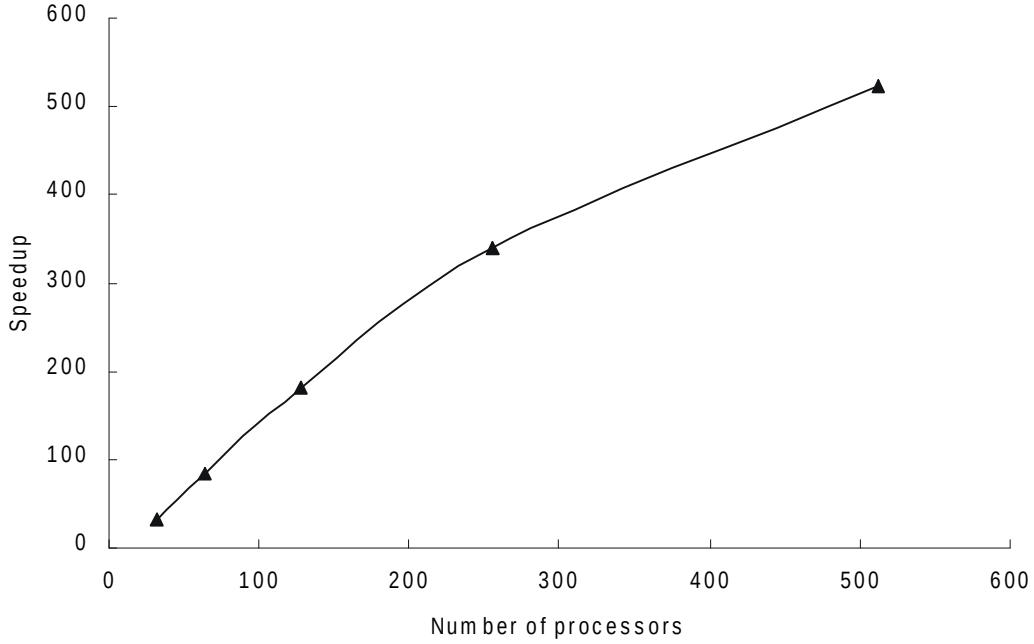


Figure 6. Speedup for the application example on the Cray T3E-900

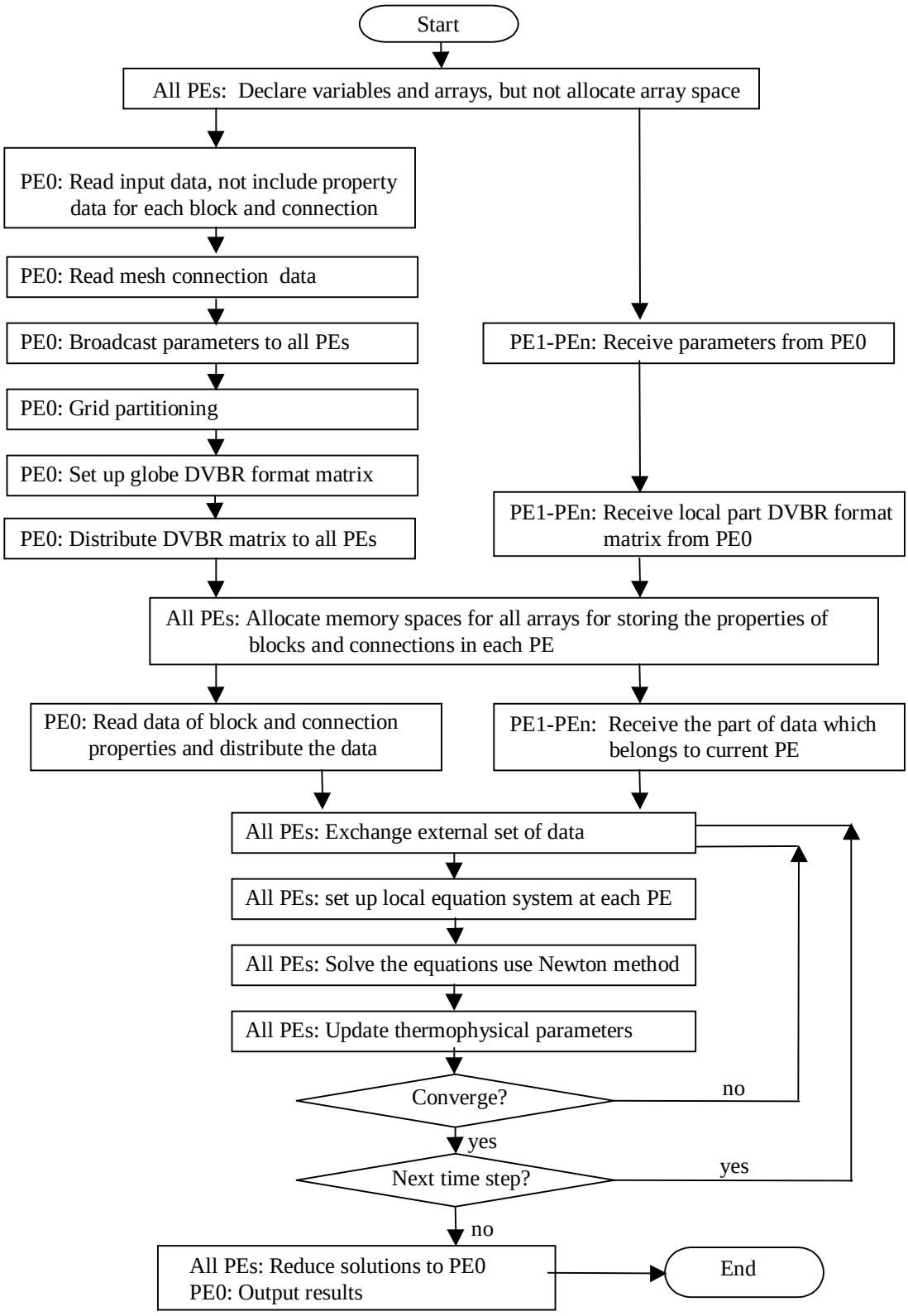


Figure 2. Simplified flow chart of parallel version TOUGH2

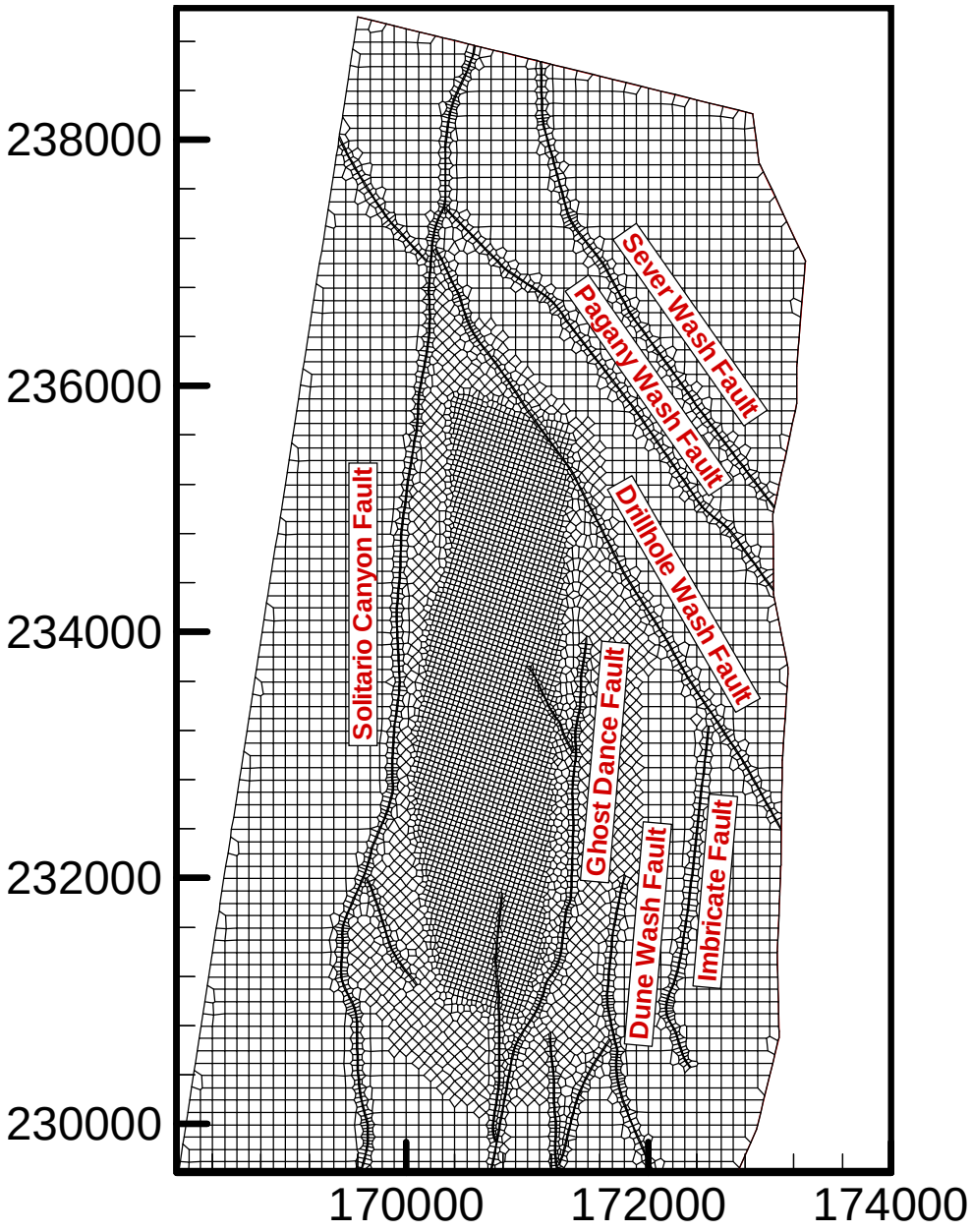


Figure 3 Plan view of the 3D simulation domain, gird and incorporated major faults

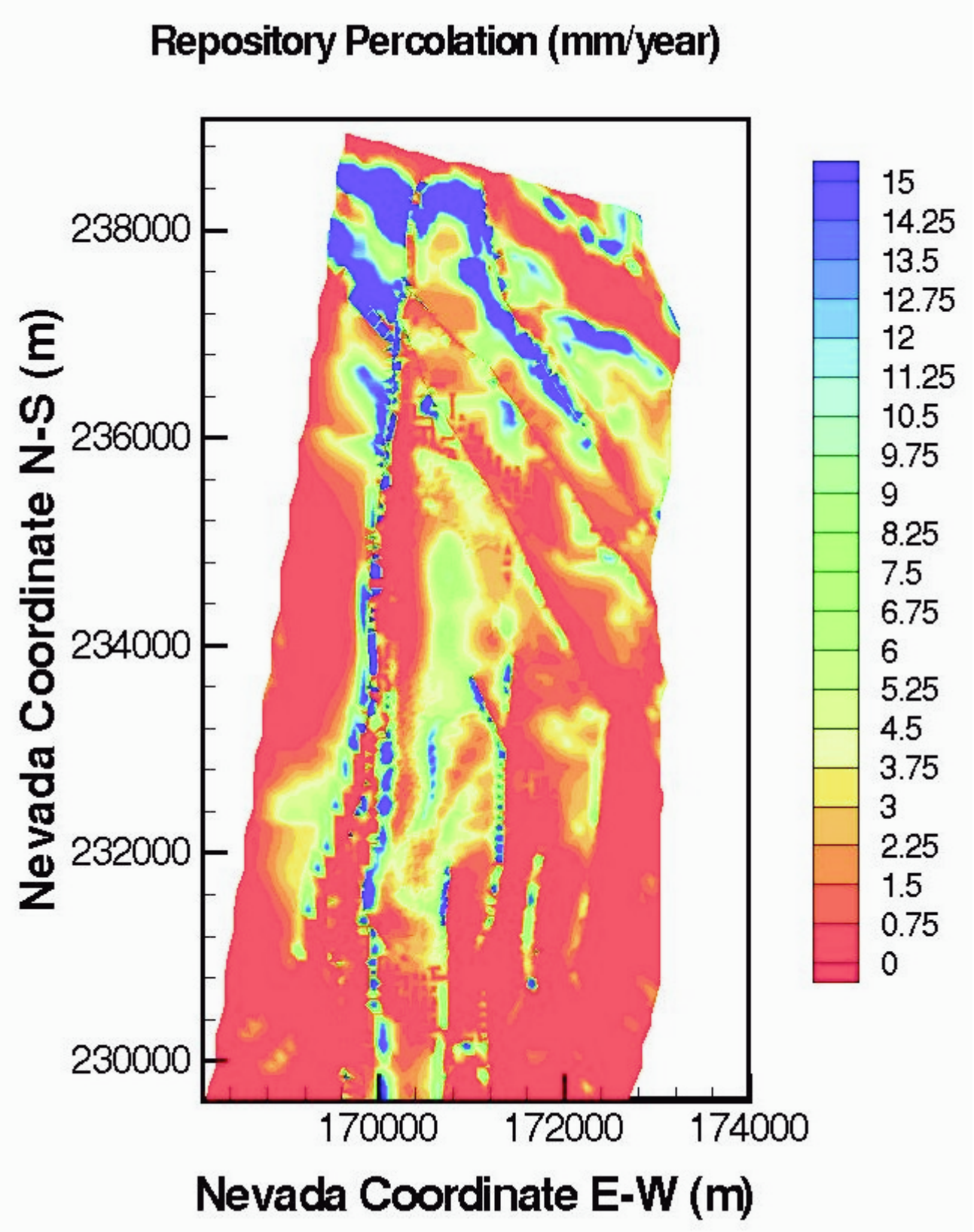
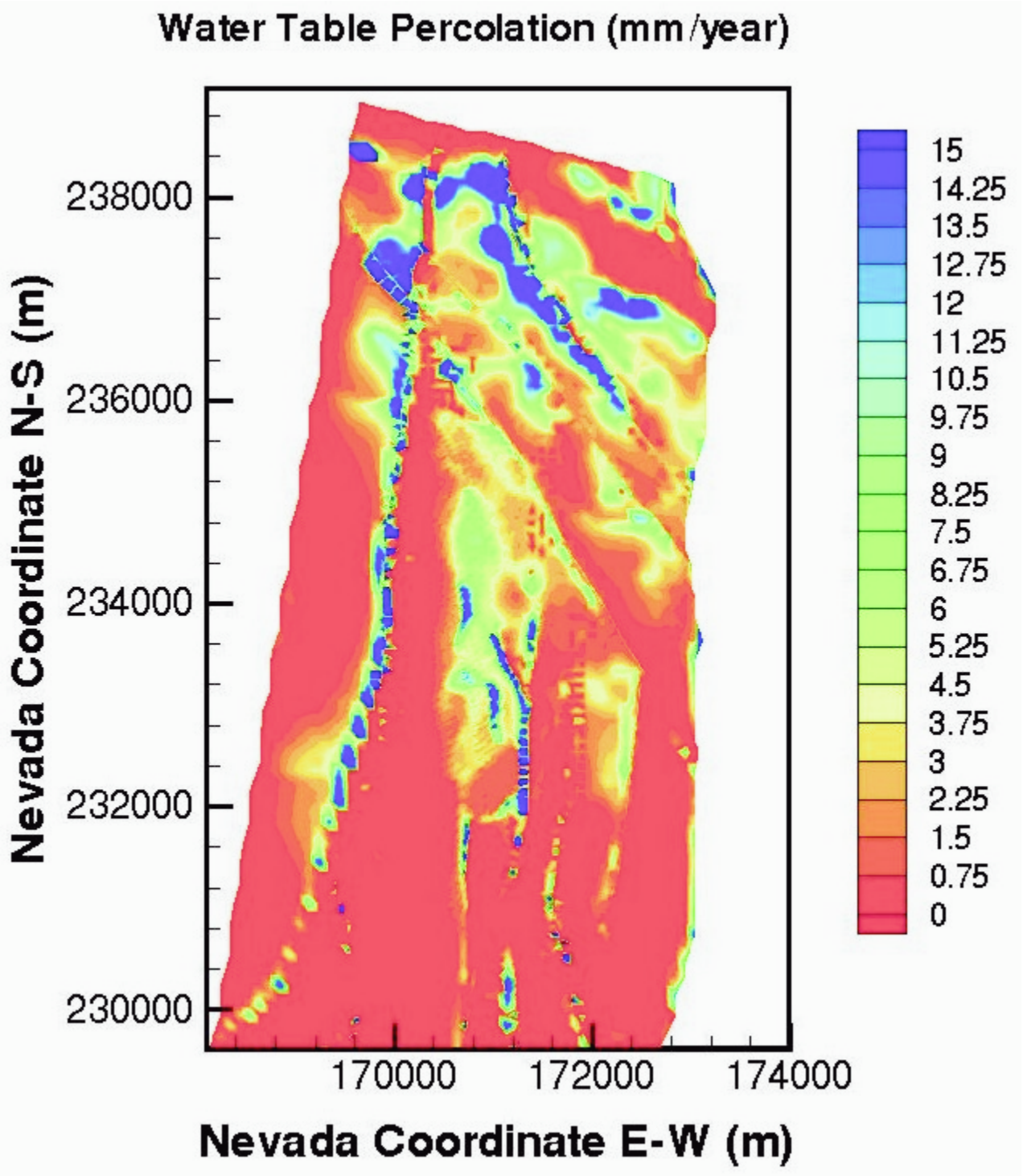


Figure 4 Simulated percolation fluxes at repository horizon


Water Table Percolation (mm/year)
Nevada Coordinate N-S (m)
238000
236000
234000
232000
230000
170000
172000
174000
Nevada Coordinate E-W (m)
15
14.25
13.5
12.75
12
11.25
10.5
9.75
9
8.25
7.5
6.75
6
5.25
4.5
3.75
3
2.25
1.5
0.75
0