A Chemical Compositional Reservoir Simulator
on Distributed Memory Parallel Computers:
Comparative Parallel-UTCHEM Simulation
Performance Study (Part I)

M. Ramé
M. Delshad

November 1994

TR94-41
A Chemical Compositional Reservoir Simulator on Distributed Memory Parallel Computers: Comparative Parallel-UTCHEM Simulation Performance Study (Part I)
M. Ramé† and M. Delshad*

†Department of Computational and Applied Mathematics, Rice University, P.O. Box 1852, Houston, TX 77251
*Department of Petroleum Engineering
The University of Texas, 26th and Speedway, CPE 2.502, Austin, TX 78712

November 16, 1994

ABSTRACT

This paper presents the application of distributed memory parallel computers to field scale reservoir simulations using a parallel version of UTCHEM, The University of Texas Chemical Flooding Simulator. The model is a general purpose highly vectorized chemical compositional simulator that can simulate a wide range of displacement processes at both field and laboratory scales. The original simulator was modified to run on both distributed memory parallel machines (Intel iPSC/860 and Delta, Connection Machine 5, Kendall Square 1 and 2, and CRAY T3D) and a cluster of workstations.

A domain decomposition approach has been taken towards parallelization of the code. A portion of the discrete reservoir model is assigned to each processor by a set-up routine that attempts a data layout as even as possible from the load-balance standpoint. Each of these subdomains is extended so that data can be shared between adjacent processors for stencil computation. The added routines that make parallel execution possible are written in a modular fashion that makes the porting to new parallel platforms straight forward.

Results of the distributed memory computing performance of Parallel simulator are presented for field scale applications such as tracer flood and polymer flood. A comparison of the wall-clock times for same problems on a vector supercomputer is also presented.

INTRODUCTION

Problems of current importance in oil recovery processes involve transport of many species in heterogeneous media. An accurate numerical modeling of these processes requires a large scale simulation. Depending on the size of the reservoir and the data, there is a need to use up to millions of gridblocks in the simulation of oil reservoirs to adequately represent the complex geological and geophysical data now available from 3D seismic and other state of art
sources. The biggest limitation of most compositional reservoir simulators is the large computational time as well as computing memory required to simulate very large scale reservoir problems with fine gridblocks.

High performance vector computers of Cray-type can sometimes handle these problems, though at a prohibitive cost. Over the last ten years, the rapid development of distributed memory parallel computers has appeared to offer the required high performance at a moderate cost for large-scale reservoir simulation\footnote{1}. As a consequence, there has been an increasing interest in parallel computing using the compositional reservoir simulators during the last few years due to its potential to model large and complex problems faster and more economically\footnote{2-4}. 

An approach of data decomposition by subdomains was taken in porting the serial version of the UTCHEM simulator to a collection of distributed memory parallel processors. The targeted systems are the Intel iPSC/860 (hypercube) and Touchstone Delta, the Thinking Machines Connection Machine 5, a heterogeneous cluster of workstations using PVM, the Kendall Square 1 and 2 series, and the CRAY T3D. This paper presents results on the Intel and Thinking Machines systems only. The remaining systems will be addressed at a later time although the code has been tested and validated against the serial simulator on all systems mentioned above. A description of each of the parallel computers used in this work is given. We also compared the performance of the parallel simulator on different machines using field scale polymer and tracer flooding examples.

One of the major bottlenecks in parallelizing reservoir simulators is in the treatment of linearized finite difference equations addressed by many authors\footnote{1,3,5-7}. A new domain decomposition solver based on additive Schwarz preconditioning\footnote{8,9} of the conjugate gradient algorithm has been added to the simulator to achieve a better parallel performance.

**PARALLEL COMPUTING**

Parallel (or distributed) computing refers to the partition of a large problem into smaller pieces so that a number of processors can concurrently effect the computations. The concept behind this approach is that concurrent computing can substantially reduce the computing time as compared to the time necessary to run the same application on a single processor of similar capabilities to one of the members of the concurrent processor set.

Parallel computers belong to one of two families. Those in which all processors can directly access data on any memory bank are called shared memory parallel processors. These machines typically have a few (4 to 16) rather powerful processors on the network, which are directly connected to each one of several memory banks. Examples of this kind are the CRAY
Y-MP and the IBM 3090VF supercomputers. The cost of building these systems is understandably high, since they require a large number of interconnections between the various processors and all of the memory banks.

Distributed memory parallel computers have an array of processors (typically less powerful than those in the above family), each of them with its own computing memory. Examples of this type are the CM-5, the Intel iPSC/860, Delta and Paragon, the CRAY T3D, and the Kendall Square 1 and 2 (the last two can run in either the shared or the distributed memory model). Data are therefore distributed across the processors and each processor can only access directly a portion of the global data. If required by the computation, processors have to exchange data. This idea will be further explained in the next section, in connection with the actual parallelization of the simulator. For now, though, one can think of the parallel simulation as spending a fraction of the total elapsed time on actual computations, \( t_{\text{CPU}} \), and a fraction on message-passing operations, \( t_{\text{mp}} \), to share the necessary data. The computing time, \( t_{\text{CPU}} \), corresponds to the total execution time of the serial algorithm. In parallel the total execution time is given by

\[
t_{\text{par}} = \frac{t_{\text{CPU}}}{N_{\text{CPU}}} + t_{\text{mp}},
\]

where \( N_{\text{CPU}} \) is the number of processors in the array. In the above equation, it is assumed that the entire algorithm can be executed in parallel. The speed-up for the algorithm is therefore given by \( \frac{t_{\text{CPU}}}{t_{\text{par}}} \), i.e.,

\[
\text{Speed-up} = \frac{N_{\text{CPU}}}{1 + \frac{t_{\text{mp}}}{t_{\text{CPU}}} \frac{N_{\text{CPU}}}{t_{\text{CPU}}}}.
\]

The above equation is known as Amdhal's law and shows that the theoretical acceleration by parallel computing is limited by the message-passing overhead time, \( t_{\text{mp}} \). When this time approaches zero, the speed-up approaches \( N_{\text{CPU}} \).

The relative message-passing overhead of an algorithm is given by the ratio \( \frac{t_{\text{mp}}}{t_{\text{CPU}}} \) which can be estimated from the parallel speed-ups and the number of processors. The maximum attainable speed-up approaches \( \frac{t_{\text{CPU}}}{t_{\text{mp}}} \) for a large \( N_{\text{CPU}} \), independent of the actual value of \( N_{\text{CPU}} \). This shows that to obtain a significant performance, it is crucial to minimize the message-passing overhead of a parallel algorithm.

The UTCHEM simulator was ported to three distributed memory parallel processors the Intel iPSC/860 (hypercube) and Touchstone Delta and the Thinking Machines Connection Machine 5. In this work, we show results on the CM-5 and the Touchstone Delta. The iPSC/860 is fully compatible with the Delta as far as running code but is an outdated model. Consequently, no results are shown for this system although the code has been extensively
tested on it. Additionally, UTCHEM has been ported to the Kendall Square 1 and 2. The emphasis of this work was the inclusion of a parallel interactive visualization package and these results will be presented in a separate publication in the near future. Finally, two PVM implementations of the parallel simulator were recently completed, one for the CRAY T3D and the other for a heterogeneous cluster of workstations. Some preliminary runs have been made on both PVM-based platforms, but more detailed results will be presented at a later time.

**Hypercube iPSC/860 Architecture**

An iPSC/860 hypercube is a parallel processor configurable up to 128 computing nodes. The processor is RISC-based Intel i860 chip with a peak computing rate of 20 Mflops in double precision and with 8 to 16 Mbytes of memory. Due to compiler limitations, there is only 2-4 Mflops for common applications. A compute node can have up to 6 nearest neighbor direct connections. The system's name comes from the fact that these are cubes of dimension greater than 3. The hypercube connects with the outside world through a front end of PC-386 type. There are parallel storage devices on the hypercube, as well as serial storage devices on the front end. Node level languages are Fortran 77 and C. These languages include NX message-passing library extensions.

**Touchstone Delta Architecture**

The Touchstone Delta computer is a prototype (one of a kind) massively parallel processor that can be configured up to 516 computing nodes. The compute nodes are improved i860 chips, with a peak rating of 60 Mflops in double precision. Compiler peculiarities limit the expected performance to around 5-7 Mflops. Each compute node has 16 Mbytes of memory, and is connected to a message routing chip (MRC). The MRC's are components of a two-dimensional mesh-type communication network, so that each CPU on the Delta has only four nearest neighbors. The system supports node level code in Fortran 77 and C, with the message-passing extensions, using the NX library as for the hypercube. The communication software makes no attempt of optimizing the routing and messages proceed always in one direction of the mesh first and, then, in the perpendicular direction.

**Connection Machine Architecture**

A Connection Machine CM5 is a massively parallel, distributed memory computer. It may be configured with tens to thousands of processing nodes. Each node consists of a SPARC scalar chip set, four vector pipelines, 32 Mbytes of memory, and a network interface. Nodes communicate with each other and with a variety of I/O devices via a point to point data routing network (DR) and a multifunction broadcast, combining, reduction network (CN). The CM5 may be programmed in the message passing style using the CMMD message passing
library. This supports node level code written in Fortran 77 or C. A CM5 may also be programmed in the data parallel style using either CM Fortran, a Fortran 90 variant, or C*, a parallel extension of the C language. The data parallel languages may be used to construct programs that are global in extent (they manage the resources of all processors in the system) or they may be used to create node level programs which communicate using the message passing library. Currently, the vector pipelines may be accessed either by writing data parallel code or by writing assembly level routines embedded in message passing programs. Node level programs written in Fortran 77 will not be able to take advantage of the vector performance of the CM5 without the addition of such assembly level routines. A well structured Fortran 77 program may be converted to CM Fortran, however by using the CMax conversion tool. This is provided as a standard component of the CM5 software suite.

**MODEL DESCRIPTION**

**The Serial Simulator**

UTCHEM is a multicomponent, multiphase, three-dimensional compositional with variable temperature reservoir simulation model\(^\text{10-12}\). The basic dynamic equations are as follows:

1. The mass balance equations which are solved for up to 21 species.
2. The aqueous phase pressure is obtained by an overall mass balance on volume occupying species (water, oil, surfactant, alcohol, and gas). The other phase pressures are computed by adding the capillary pressure between phases.
3. The energy balance equation which includes heat flow between the reservoir and the over- and underburden rocks.

Four phases are modeled. The phases are a single component gas phase \((\ell=4)\) and up to three liquid phases: aqueous \((\ell=1)\), oleic \((\ell=2)\), and microemulsion \((\ell=3)\) depending on the relative amounts and effective electrolyte concentration (salinity) of the surfactant/oil/water phase environment.

The flow equations allow for compressibility of rock and fluids, dispersion and molecular diffusion, chemical reactions, and phase behavior and are complemented by constitutive relations.

The model includes options for multiple wells completed either horizontally or vertically. Aquifer boundaries are modeled as constant potential surfaces or as closed surfaces.

The resulting flow equations are solved using a block-centered finite-difference scheme. The solution method is implicit in pressure and explicit in concentration (IMPES-like). Either One, two-point upstream, or third-order spatial discretization are used. To
increase the stability and robustness of the third-order method, a flux limiter that is total-variation-diminishing (TVD) has been added\textsuperscript{13,14}.

UTCHEM has been widely used to simulate both laboratory and field scale processes such as water, polymer, surfactant/polymer, profile control using gel, high pH alkaline/surfactant/polymer, single well or interwell tracer. The code has recently been modified for applications in groundwater contamination and remediation.

**Mass Conservation Equations:** The assumptions imposed when developing the flow equations are 1) local thermodynamic equilibrium except for tracers, 2) immobile solid phases, 3) slightly compressible rock and fluids, 4) Fickian dispersion, 5) ideal mixing, and 6) Darcy's law. The boundary conditions are no flow and no dispersive flux across the impermeable boundaries.

The continuity of mass for component $\kappa$ in association with Darcy's law is expressed in terms of overall volume of component $k$ per unit pore volume ($\tilde{C}_k$) as

$$\frac{\partial}{\partial t} \left( \phi \tilde{C}_k \rho_k \right) + \nabla \cdot \left[ \sum_{\ell=1}^{n_p} \rho_k \left( C_{k\ell} \tilde{u}_\ell - \tilde{D}_{k\ell} \right) \right] = R_k,$$

where the overall volume of component $\kappa$ per unit pore volume is the sum over all phases including the adsorbed phases:

$$\tilde{C}_k = \left( 1 - \sum_{\kappa=1}^{n_{cv}} \hat{C}_\kappa \right) \sum_{\ell=1}^{n_p} S_\ell C_{k\ell} + \hat{C}_k \quad \text{for } k = 1, \ldots, n_c .$$

$n_{cv}$ is the total number of volume occupying components. These components are water, oil, surfactant, and gas. $n_p$ is the number of phases; $\hat{C}_\kappa$ is the adsorbed concentration of species $\kappa$; and $\rho_k$ is the density of pure component $\kappa$ at a reference phase pressure $P_R$ relative to its density at reference pressure $P_{R0}$, usually taken at the surface condition of 1 atm. We assume ideal mixing and small and constant compressibilities $C_k^0$.

$$\rho_k = 1 + C_k^0 \left( P_R - P_{R0} \right)$$

The dispersion flux is given by

$$\tilde{D}_{k\ell} = \phi S_\ell \tilde{K}_{k\ell} \cdot \nabla \tilde{C}_{k\ell} .$$

The dispersion tensor ($\tilde{K}_{k\ell}$) including molecular diffusion ($D_{k\ell}$) is calculated as
follows\textsuperscript{15}:
\[
K_{K\ell ij} = \frac{D_{K\ell}}{\tau} \delta_{ij} + \frac{\alpha_{T\ell}}{S_{\ell}} \bar{u}_{\ell} \delta_{ij} + \frac{(\alpha_{L\ell} - \alpha_{T\ell})}{\phi S_{\ell}} \frac{u_{\ell i} u_{\ell j}}{\bar{u}_{\ell}}, \tag{7}
\]

where $\alpha_{L\ell}$ and $\alpha_{T\ell}$ are phase $\ell$ longitudinal and transverse dispersivities; $\tau$ is the tortuosity; $\delta_{ij}$ is the Kronecker delta function; and $u_{\ell i}$ and $u_{\ell j}$ are the component of Darcy velocity of phase $\ell$ in i and j directions. The magnitude of vector flux for each phase is computed as
\[
|\bar{u}_{\ell}| = \sqrt{(u_{x\ell})^2 + (u_{y\ell})^2 + (u_{z\ell})^2}. \tag{8}
\]

The phase flux from Darcy’s law is
\[
\bar{u}_{\ell} = -\frac{k_{r\ell}}{\mu_{\ell}} \cdot \left( \nabla p_{\ell} - \gamma_{\ell} \bar{v} h \right), \tag{9}
\]

where $\bar{k}$ is the intrinsic permeability tensor and $h$ is the vertical coordinate. $k_{r\ell}$, $\mu_{\ell}$, and $\gamma_{\ell}$ are the relative permeability, viscosity, and specific weight for phase $\ell$.

The source terms $R_{K\ell}$ are a combination of all rate terms for a particular component and may be expressed as
\[
R_{K\ell} = \phi \sum_{\ell=1}^{n_{P}} S_{\ell} r_{K\ell} + (1 - \phi) r_{Ks} + Q_{K}, \tag{10}
\]

where $Q_{K}$ is the injection/production rate for component $\kappa$ per bulk volume. $r_{K\ell}$ and $r_{Ks}$ are the reaction rates for component $\kappa$ in phase $\ell$ and solid phase $s$.

**Energy Conservation Equation:** The energy balance equation given below is derived by assuming that energy is a function of temperature only and energy flux in the reservoir occurs by heat conduction and convection flow only.
\[
\frac{\partial}{\partial t} \left[ (1 - \phi) \rho_s C_v s + \phi \sum_{\ell=1}^{n_p} \rho_\ell S_\ell C_v \ell \right] T + \\
\vec{v} \cdot \left( \sum_{\ell=1}^{n_p} \rho_\ell C_p \ell u_\ell T - \lambda_T \vec{v} \cdot T \right) = q_H + q_L
\]

where \(T\) is the reservoir temperature; \(C_{vs}\) and \(C_{v\ell}\) are the reservoir rock and phase \(\ell\) heat capacities at constant volume; \(C_{p\ell}\) is the phase \(\ell\) heat capacity at constant pressure; and \(\lambda_T\) is the thermal conductivity (all assumed constant). \(q_H\) is the enthalpy source term per bulk volume. \(q_L\) is the heat loss to overburden and underburden rocks computed using the Vinsome and Westerveld\textsuperscript{16} heat loss method.

**Pressure Equation:** The pressure equation is developed by summing the mass balance equations over all volume occupying components, substituting Darcy's law for the phase flux terms, using the definition of capillary pressure, and noting that \(\sum_{k=1}^{n_{cv}} C_{k\ell} = 1\). The pressure equation in terms of the reference phase pressure (phase 1) is

\[
\phi C_t \frac{\partial P_1}{\partial t} + \vec{v} \cdot \nabla P_1 = -\nabla \cdot \left( \sum_{\ell=1}^{n_p} k_\ell \lambda_{r\ell c} \vec{v} P_1 \right) \\
+ \sum_{\ell=1}^{n_p} k_\ell \lambda_{r\ell c} \vec{v} P_{c\ell 1} + \sum_{k=1}^{n_{cv}} Q_k
\]

where \(\lambda_{r\ell c} = \frac{k_{r\ell c}}{\mu_\ell} \sum_{k=1}^{n_{cv}} \rho_k C_{k\ell}\) and total relative mobility with the correction for fluid compressibility is \(\lambda_{rTc} = \sum_{\ell=1}^{n_p} \lambda_{r\ell c}\).

The total compressibility, \(C_t\), is the volume weighted sum of matrix (\(C_r\)) and component compressibilities (\(C_k\)):

\[
C_t = C_r + \sum_{k=1}^{n_{cv}} C_k^0 \tilde{C}_k,
\]

where \(\phi = \phi_R \left[ 1 + C_r (P_R - P_{R0}) \right]\).
The Parallel Simulator

Systems with as flexible a computing model as those described above offer several different strategies for porting the UTCHEM simulator. We have identified two options that do not require completely rewriting the code. In the first, an independent program is executed on each node. In the second, a problem is distributed across all the nodes of the machine in a message passing model.

The first approach required modest modification to the serial code to allow it to execute independently on each node of the processor array. This port was tried on the CM-5 only. In this instance, the code required only a few small changes to ensure that I/O was handled properly within the parallel message passing environment. No message passing calls or data mapping changes were introduced. When executed, this code runs a copy of the program with each node computing a different problem. This port required only a few hours to complete. We note that while the port was quickly done, there was no computational speed up introduced. The ability to run many simulations in parallel can find use in conditional simulation, where different geostatistical realizations of rock properties can be used as input. This use of the parallel port has not been pursued at this point.

Our second port, using message passing, was designed to use all of the computing nodes in the simulation of a single problem by the single-program-multiple-data (SPMD) programming model. Upon execution, this version of the code distributes the spatial gridblocks across all of the processors. Each processor will generally receive many blocks. Array declarations were changed to allocate only the required memory to work on the subdomain of the reservoir contained in each processor. However, since the majority of arrays are involved in a stencil calculation (arising from the discretization of the model equations) all of the arrays were declared to allocate enough memory for a subdomain plus a three-dimensional padding envelope around the subdomain so that data from neighboring processors can easily be brought into a given processor. By this strategy, the long-vector style of the original serial code can be easily preserved and, therefore, the parallel code can take advantage of machines with a node-level vectorizing Fortran 77 compiler.

The gridblock distribution scheme assigns consecutively numbered blocks (for some numbering of the blocks in the system) to the nodes of the machine. This results in reasonable locality of computation (nearest neighbors are frequently located on the same computing node), and reasonable load balance is achieved (blocks are assigned so that nodes receive either \( \frac{\text{NBL}(x, y, z)}{\text{N_CPU}(x, y, z)} \) or \( \frac{\text{NBL}(x, y, z)}{\text{N_CPU}(x, y, z)} + 1 \) gridblocks, where \( \text{NBL}(x, y, z) \) is the global number of gridblocks in the x, y, or z directions, respectively, and \( \text{N_CPU}(x, y, z) \) is the number of processors into which the problem is decomposed in the x, y, or z direction, respectively).
Figure 1a shows a two-dimensional example of the gridblock distribution strategy for nine processors. Figure 1b shows the size of the array declared on processor 4 (processors are numbered sequentially from 0 through 8), where both global and local numbering schemes can easily be identified. Computations on the blocks are then performed in parallel, where each processor performs the same computations on a subset of the global problem (SPMD model) in approximate synchronization. Data between neighboring processors are communicated at specified synchronization points by explicit message passing calls.

In this port, the I/O was handled serially through a designated node in the processor array. This was done to facilitate portability of the code and amounts to all processors sending their partial output to the collecting processor and, subsequently, the collecting processor writing to a disk. In principle this could be handled by declaring one global array (big enough to hold one entry per gridblock of the global problem). However, because of the way the I/O was handled in the original serial code and since the aim of this work was to not change any features of the simulator from the users standpoint (e.g., the structure of the output files), a global array big enough to hold 25 times the global number of gridblocks had to be declared. Since the memory per processor of distributed memory machines is relatively small and the SPMD programming model is used, i.e., each processor loads the same code on its memory, this imposes severe limitations on the maximum size of the problem that could be run. This poor I/O handling can be improved and work is under way to minimize and ultimately eliminate this adverse effect. However, in the absence of a standard for parallel I/O across parallel computers, it is our opinion that no significant amount of developing time should be devoted to the handling of I/O until a particular parallel machine has been chosen as a target.

The message passing was first implemented for the Intel systems using the NX library environment. Twelve routines were added to the original 31 in the process of parallelizing the code. The communication is done synchronously in this implementation, meaning that no computation is being performed while the processors exchange data. General code modifications were also necessary since not all gridblocks on the boundary of a subproblem are boundary gridblocks for the global problem.

Since CMMD, the CM5 message passing environment, is a superset of the functions found in the Intel NX environment, converting the code to run on the CM5 required only that we provide emulation functions for the Intel primitives and small modifications to account for differences in I/O behavior. This version of UTCHENM used only 18 communications related functions. Emulations for these NX calls were created with minor effort.

Both coarse (subdomain) and fine (do-loop) grain parallelism were identified in this code. The original (serial) simulator was written to perform in a near-optimal way on vector supercomputers, i.e., the data structures are declared as long vectors where each entry refers to
a physical \((x, y, z)\) location in the reservoir by means of a given numbering scheme for the gridblocks. These data structures are not naturally suited for parallel computing in the message-passing model. The dilemma in parallelizing do loops appeared as follows: if one wants to keep the vector style computation, in order to keep the code modifications to a minimum and to make use of vectorization on the processors of some parallel systems, a significant amount of unnecessary computing overhead is created by do loops that sweep over the portions of the arrays that lie on the subdomain extension, and not in its interior. The computed values for these gridblocks in the *padded region* are incorrect and have to be replaced every time by the correct values for them known by the neighboring processors (i.e., by interprocessor communication). The first approach to do-loop parallelization was, nevertheless, exactly this. For instance, an original serial loop that looks like

```
do 10 i = 2, nbl
    a(i) = b(i-1) * c(i)
10  continue
```

was initially parallelized as follows,

```
call pad (1, 0, 0, b)
call ploop (2, nbl, 1, il, ih, inc )
do 10 i = il, ih
    a(i) = b(i-1) * c(i)
10  continue
```

Subroutine *pad* is a multi-directional message-passing routine, where each of the first three (integer) arguments define the width of required padding (in number of gridblocks) and the real argument is the array to be padded. In the example above, array \(b\) is padded in the \(x\)-direction only by a single slice of gridblocks. Additionally, processors on the \(x=0\) boundary of the reservoir have to correct the values of array \(b\) on the opposite subdomain interface, much like the serial code does it on the \(x=x_{\text{max}}\) boundary. Subroutine *ploop* finds, for a given low bound, high bound and increment of a serial do loop, the processor-dependent low and high bounds and increment for the parallel loops. Looking back at Figure 1, *ploop* would give, for processor 4, \(il=2, ih=29\), and \(inc=1\). Obviously a large number of grid blocks on the subdomain extension or padding are included in the resulting parallel loop.

A further modification was introduced to eliminate this overhead. Each do-loop was split up into a triply (at most) nested loop, each nesting level referring to one coordinate
direction of the 3-D domain, to produce the following

call pad (1, 0, 0, b)
if ( node . eq . 0 ) then
    i3 = nzl
    i2 = nyl
    do i1 = nxl + 1, nxh
        i = ijkposgl(i1, i2, i3)
        a(i) = b(i-1) * c(i)
    enddo
    i3 = nzl
    do i2 = nyl + 1, nyh
        do i1 = nxl, nxh
            i = ijkposgl(i1, i2, i3)
            a(i) = b(i-1) * c(i)
        enddo
    enddo
    do i3 = nzl + 1, nzh
        do i2 = nyl, nyh
            do i1 = nxl, nxh
                i = ijkposgl(i1, i2, i3)
                a(i) = b(i-1) * c(i)
            enddo
        enddo
    enddo
else
    do i3 = nzl, nzh
        do i2 = nyl, nyh
            do i1 = nxl, nxh
                i = ijkposgl(i1, i2, i3)
                a(i) = b(i-1) * c(i)
            enddo
        enddo
    enddo
endif
Clearly in this case, the code modification proved extensive because of the guards that have to be introduced and the dissection of loops necessary within some branches of the guard. In the example above, node is the processor number. Processor 0 is the only particular case in the example above because of the shift in array b. Therefore, the computation for this processor was split up into three sections: an edge minus one gridblock, the top face minus one edge and the volume minus the top face. The net result is that only one gridblock of processor 0 is unaffected by this computation. All other processors get the desired result in just one pass of the triply-nested do-loop construct. Other, more involved vector-style do-loops generated greater source code explosions as a result of this transformation. Function ijkposgl(i1, i2, i3) returns the local long-vector counter for the array position in each subdomain corresponding to the x-, y- and z-directional global indices, i.e., i1, i2, and i3 respectively, which range between 1 and the global number of gridblocks in each coordinate direction of the reservoir. This last phase of do-loop conversion produced a code that approaches linear speed-up as the subdomain surface-to-volume ratios decrease.

Coarser grain parallelism was improved by extensive algorithmic changes. The existing linear solver in the serial code, a Jacobi-preconditioned Conjugate Gradient scheme (JCG), did not scale well in our preliminary tests. Therefore, an Parallel Overlapping Additive Schwarz preconditioned Conjugate Gradient algorithm (POAS) was added as an option for the iterative solution of the aqueous phase pressure linear system. The work per subdomain involved in the POAS solver is substantially greater than that in the JCG. However, POAS method is much more robust and has excellent scaling characteristics.

The handling of communication, global operations and other machine dependent calls was designed in a way that makes the code easily portable. Parallelized original UTCHEM routines that require communication call routines that contain system dependent library calls. These routines (wrappers) were written for Intel, TMC, KSR, T3D, and heterogeneous PVM so that, at compile time, different wrappers are linked depending on the target machine. In its present state, the code can be run on all the above mentioned systems by invoking the makefile that creates executable for the machine of choice.

**DESCRIPTION OF EXAMPLE DATA**

**Tracer Flood**

The first example studied using the parallel code was based on a multiple-well tracer field project conducted by Oryx Energy Co.. The 320 acre area of interest included 13 producing wells and 4 injectors in which 7 different tracers were injected. A 31x45x3 grid was used with each gridblock size on the order of 100 ft by 100 ft in area. The sandstone reservoir
was heterogeneous both areally and vertically. Therefore, the permeability in each simulation layer varied with both direction and location. The three vertical layers had different thickness and permeability. The detailed field data and tracer analysis are given in Allison et al.\textsuperscript{18}. The results of a finer mesh of 62x90x3 run on a Cray computer can be found in Pope et al.\textsuperscript{19}. The simulation performed for up to 50 days whereas the full simulation was 950 days.

**Polymer Flood**

The second example was based on a polymer flood pilot test in the Courtney sand of Chateauerenard Field in France. The performance of this pilot was simulated by Takagi\textsuperscript{20,21} using the serial version of the code. Takagi used 25x25x3 gridblocks as the finest grid in his mesh refinement study with the gridblocks still on the order of 65 ft long, 65 ft wide, and 3 ft thick. The reservoir description used in our example was a layered one and therefore the discretization sizes in both areal directions can easily be changed in order to test the code performance versus changing surface-to-volume ratios in the subdomains. The configuration of the pilot was an inverted five-spot pattern with an average distance of 756 ft between producing wells. The pilot operation consisted of successive injection of a slug of 0.1 wt% hydrolyzed polyacrylamide tapered down to concentration of 0.02 wt% in the final slug and followed by chase water injection. To study the effect of mesh refinement and to increase the number of gridblocks to take advantage of the parallelism, the number of gridblocks were increased to 50x100x6. The simulation performed for only 20 days whereas the full simulation was 1540 days. The number of species for which the conservation equation is solved for was 5 in this example.

**Surfactant/Polymer Flood**

The third example studied was a hypothetical three-dimensional surfactant/polymer flood with a quarter of five spot well pattern in a layered reservoir. The grid sizes were 96x96x2 in the x, y, and z directions, respectively. The simulation performed for up to 20 days. The material balance equation is solved for 10 components, which are water, oil, surfactant, polymer, anion, calcium, cosurfactant, and three tracers.

**PARALLEL SIMULATOR PERFORMANCE**

Parallel runs were conducted with the above mentioned data sets. All parallel tests for the polymer flood case were run for 20 days due to the limited availability of the parallel systems involved. With this maximum simulation time, initialization amounted to a fraction of 1%. Therefore, we feel confident that these timings can be extrapolated approximately linearly to greater simulation times. All runs for the tracer data set were run for 50 days, where
initialization overhead weighs even less.

Figures 2 and 3 show the elapsed times for the tracer flood runs on 4, 8, 16, 32, and 64 CM-5 and on 4, 8, 16, 32, 64 and 100 Intel Delta processors, respectively, on a log-log plot. The solid line on these plots, with a slope of -1, indicates the ideal scaling performance. These plots shows partial timings for several tasks as well as the total elapsed time for each run. Essentially, two families of tasks are present: local operations, which do not involve message passing, and stencil computations, which require message passing. The first family includes physical property calculation routines (e.g., procedures densy and viscos) in which the return values for a gridblock depend only on the physical and chemical conditions of this same grid block. The second family includes primarily the necessary computations for explicit time stepping of the mass conservation equations (e.g., procedure coneq) and the assembly of the linear system to obtain a pressure distribution at each time level (procedure solmat includes the assembly of the linear system plus its iterative solution). Routines in the first family are not expected to show any performance degradation as more processors are added to the network, but members of the second family do show some degradation depending on the level of communication overhead involved in each case.

The conjugate gradient type solvers belong to a third family of tasks which rely heavily on dot product and matrix-vector multiplies that require global data exchanges. The idea behind the use of a robust parallel preconditioner is to decrease the number of iterations, thus decreasing the incidence of dot product type operations in the total time necessary to converge to the solution. This decrease in iteration count is achieved by setting up a preconditioner that approximates the inverse of the coefficient matrix more closely than, say, a diagonal preconditioner. Naturally, in general the cost per iteration increases as does the robustness of the preconditioner.

On the CM-5 (Figure 3), all partial timings as well as the total elapsed time show an acceptable approximation to the slope of the control line. The results on the Delta (Figure 2) clearly discriminate between the two families of tasks described above. Subroutines densy and viscos scale very closely to the ideal line up the maximum number of processors. Subroutines coneq and solmat show the anticipated performance degradation. The TOTAL elapsed time for these runs appears to be controlled by the scaling of both these routines that amount to 85-90% of the total simulation time. Of these two, procedure solmat shows a worse scaling because it includes the timing for the Jacobi Conjugate Gradient solver, which is dominated by dot product and matrix-vector multiplies, due to the large number of iterations required to converge the system. Figure 4 shows the same run on the Delta, but using the Parallel Overlapping Additive Schwarz (POAS) preconditioner for the CG method. Its initial scaling (up to 32 processors) is far more favorable than that of the diagonal preconditioning.
In the transition from 32 to 64 processors, the subdomains in this domain decomposition solver are left with little computing to do and the method starts to degrade. Notice that the TOTAL elapsed time again is dominated by the behavior of solmat.

A totally different situation is displayed on these plots by subroutine wellck, which performs some computations for the wells. As more processors are added to the used partition, its elapsed time increases. This is due to the way wells are treated throughout the parallel code, i.e., all processors know the conditions of all wells at all times of the simulation and, therefore, this and other related routines are dominated by communication overhead. However, this situation does not affect the TOTAL elapsed time since these well-related routines amount to a very small fraction of the total simulation time.

Speed-ups normalized with respect to the 4-processor elapsed times, for the runs of Figures 2 and 3 above, are shown on Figures 5 and 6. Figure 5 shows that the viscosity calculation routine speeds up almost ideally up to 64 Delta processors, but those involved in the explicit time stepping suffer some degradation by communication overhead. Figure 6 shows much better speed-ups on the CM-5 than those obtained on the Intel system for all procedures. In particular, subroutine viscous shows superlinear speed-up for some numbers of processors. This can be explained by the slight randomness involved in the timing routines.

Figure 7 shows a comparison between the speed-up characteristics of the JCG linear solver and those of the POAS solver. The JCG method gets consumed by global communication as more processors are added to the computing partition. The POAS solver scales very well up to 32 processors, at which point the subdomain problems get too small due to the small global problem size and the method degrades due to message-passing overhead. The actual elapsed time for the solver did not improve by much in going from JCG to POAS for this case, on account of the relative good conditioning of the coefficient matrix for this problem, which allows JCG to converge quickly. The fact that superlinear speed-up is attained here is due to the use of direct solvers on the subdomain problems. This is explained in detail elsewhere (see Pavarino and Ramé17).

Some general comments are in order in view of the results on both these machines. The CM-5 appears to be a more balanced system, from the view point of computing power compared to interprocessor data transfer rates. Therefore, sustained scaling and speed-up are obtained up to a greater number of processors than on the Intel Delta.

Figure 8 presents the timing results for the polymer flood data set, with a discretization containing 30,000 gridblocks on a CM-5. As expected from the greater size of the discrete model, the approximation to the slope of the control line is even better than before. Subroutine density shows an anomalous behavior for the 64-processor run. This, again may be due to instabilities of the elapsed time counter which is sporadically observed on the CM-5. The
improved performance can be more readily observed on Figure 9 (where the 64-node density reading has been deleted), for the corresponding speed-ups which are surprisingly close to the ideal solid line. These results point to the relatively slow computing speed of the Sun microprocessor on each CM-5 node. We believe that this machine could use a processor twice as fast and still show a good balance between computation and data transfer rates. Since CM-5 processors come with at least 32 Mbytes of RAM, problems with a favorable surface-volume ratio can be loaded with no problem.

The more favorable of the Intel products available to us to run this case is the Delta because of the large number of processors, considering that Intel machines are rarely found to exceed 16 Mbytes per processor, of which only about 12 are available for computing. Preliminary calculation, which was further checked experimentally showed that this case would require 256 i860 processors. Unfortunately, what we believe to be either hardware problems or faulty compiler optimization for the Delta, prevented us for running this case successfully. A series of strange run time errors was obtained when trying to run UTCHEM on more than 100 Delta processors, none of which could be traced back to coding errors. Besides, the same source code has been run on 128-processor CM-5 systems with no problem in the past.

Lastly, a surfactant/polymer example, with a 96x96x2 discretization size was successfully run on the Intel Delta, on a 10x10 partition. The elapsed time for 20 days of simulation was 27 min 49 sec. for the JCG solver and only slightly faster for the POAS solver. This minor improvement is possibly due to the growing condition number of the coefficient matrix of the pressure linear system, as the number of gridblocks increases. The difference of a few seconds is not important but the POAS solver will find good use in the case of more heterogeneous data sets.

In order to give a better perspective of the status of parallel machines, Table 1 shows a total elapsed time comparison for the cases presented in this work with respect to a single CRAY Y-MP head. Timings for the parallel systems are given for the number of processors that showed the lowest total elapsed time. Both parallel systems fall short of the CRAY performance for cases run in this work. This is, to some extent, due to the untuned nature of the code for any of these machines. The surfactant/polymer case on 100 Delta processors is the one that gets closest to the Cray time because the problem was decomposed areally into 10x10 processors and the two vertical layers generate a moderate communication overhead.

**SUMMARY AND CONCLUSIONS**

Our experience with the UTCHEM compositional simulator and the distributed memory parallel environments suggests that the conversion from a strictly serial implementation to one
which uses the full resources of the machine is an achievable, although nontrivial, task. Often, a good degree of speedup can be attained by distributing a single problem across all of the nodes of the machine.

Data structures that are natural in vector computing pose a performance problem in distributed computing, unless the code is altered fairly extensively.

The main lesson of this work has to do with the balance of parallel computing systems, i.e., whether they can sustain high parallel efficiencies for a problem of fixed size as the number of processors used increases to a large number. The amount of RAM per processor improves the machine balance since a larger subdomain problem can be loaded on each processor thus reducing the surface-to-volume ratio.

Handling I/O through a designated processor ensures portability but severely limits the maximum size of the problem that can be run, particularly, with a code like UTCHEM that models complex chemical processes.

Domain decomposition methods appear as a very efficient parallel approach to the solution of large linear systems. The results of the parallel additive Schwarz method show that one can tackle large problems with a conceptually simple solver.

**NOMENCLATURE**

\[
\begin{align*}
C_\kappa & = \text{Overall concentration of species } \kappa \text{ in the mobile phases} \\
C_\kappa^0 & = \text{Compressibility of species } \kappa \\
\hat{C}_\kappa & = \text{Adsorbed concentration of species } \kappa \\
\tilde{C}_\kappa & = \text{Overall concentration of species } \kappa \text{ in the mobile and stationary phases} \\
C_{\kappa l} & = \text{Concentration of species } \kappa \text{ in phase } l \\
C_{p\ell} & = \text{Heat capacity at constant pressure for phase } \ell \\
C_r & = \text{Rock compressibility} \\
C_T & = \text{Total compressibility} \\
C_{v\ell} & = \text{Heat capacity at constant volume for phase } \ell \\
D_{\kappa l} & = \text{Diffusion coefficient of species } \kappa \text{ in phase } l \\
h & = \text{Thickness} \\
K & = \text{Dispersion coefficient} \\
\tilde{K} & = \text{Permeability tensor} \\
k & = \text{Soil permeability} \\
k_{r\ell} & = \text{Relative permeability of phase } \ell \\
P_{C\ell\ell'} & = \text{Capillary pressure between phases } \ell \text{ and } \ell' \text{.} \\
P_\ell & = \text{Pressure of phase } \ell
\end{align*}
\]
\( P_R \) = Reference pressure
\( Q_{\kappa} \) = Unit volumetric Source/sink for species \( \kappa \)
\( Q_L \) = Heatloss to surrounding rocks
\( q_h \) = Unit enthalpy Source term
\( S_\ell \) = Saturation of phase \( \ell \)
\( T \) = Temperature
\( t \) = Time
\( \bar{u}_\ell \) = Darcy flux

**Greek Symbols**

\( \alpha_L \) = Longitudinal dispersivity
\( \alpha_T \) = Transverse dispersivity
\( \gamma_{\kappa R} \) = Specific weight of species \( \kappa \) at reference pressure
\( \mu_\ell \) = Viscosity of phase \( \ell \)
\( \lambda_\ell \) = Relative mobility of phase \( \ell \)
\( \lambda_{RT} \) = Total relative mobility
\( \lambda_T \) = Thermal conductivity
\( \rho_\ell \) = Density of phase \( \ell \)
\( \phi \) = Porosity
\( \Phi \) = Potential
\( \tau \) = Tortuosity

**Subscripts**

\( \kappa \) = Species number
1 = Water
2 = Oil
3 = Surfactant
4 = Polymer
5 = Chloride
6 = Calcium
7 = Alcohol 1
8 = Gas
9 = Tracer 1
10 = Tracer 2
11 = Tracer 3

For Gel option:
12 = Sodium dichromate (Na2Cr2O7)
13 = Thiourea (CSN2H4)
14 = Trivalent chromium (Cr³⁺)
15 = Gel
16 = Hydrogen
For Geochemistry option:
12 = Sodium
13 = Hydrogen
14 = Magnesium
15 = Carbonate
16 = Acid species of crude oil or aluminum
17 = Silica

ℓ = Phase number
1 = Aqueous
2 = Oleic
3 = Microemulsion
4 = Gas

r = Residual
s = Solid

ACKNOWLEDGMENTS
The financial support from U.S. Department of Energy, participating companies of the Enhanced Oil and Gas Recovery Research Program of the Center for Petroleum and Geosystems Engineering at the U. of Texas-Austin, and participating companies of the Subsurface Modeling Group at Rice University is acknowledged. The authors would also like to thank Hector Klié from Rice University for assistance in PVM conversion of the code. Special appreciation goes to Adam Greenberg from Thinking Machines Corp. and Kirk Jordan from Kendall Square for unlimited help and guidance in this research. We also like to thank Thinking Machines Corp. and Kendall Square for providing the computing resources for this work.

REFERENCES


Table 1: A comparison of the elapsed time for different machines

<table>
<thead>
<tr>
<th>Run</th>
<th>CRAY Y-MP (Minutes)</th>
<th>CM-5 with 64 nodes (Minutes)</th>
<th>Delta with 100 nodes (Minutes)</th>
<th>Grid size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracer</td>
<td>2</td>
<td>57</td>
<td>20</td>
<td>31x45x3</td>
</tr>
<tr>
<td>Polymer</td>
<td>3</td>
<td>32</td>
<td>---</td>
<td>50x100x6</td>
</tr>
<tr>
<td>Surf./polymer</td>
<td>11</td>
<td>---</td>
<td>27</td>
<td>96x96x2</td>
</tr>
</tbody>
</table>

(a)

(b)

Figure 1: Local and global numbering schemes for the parallel code. (a) Mapping of a two-dimensional problem onto an array of nine processors (numbered 0 through 8). Thick lines indicate interprocessor boundaries. (b) Local and global grid block numbers appear on upper right corner and lower left corner of each grid block, respectively, for the process loaded on processor 4.
Figure 2: Elapsed times for tracer flood on the Intel Delta using JCG solver.

Figure 3: Elapsed times for tracer flood on the CM-5 using JCG solver.

Figure 4: Elapsed times for tracer flood on the Intel Delta using POAS solver.

Figure 5: Parallel speed-ups for tracer flood on the Intel Delta using JCG solver.
Figure 6: Parallel speed-ups for tracer flood on the CM-5 using JCG solver.

Figure 7: Comparison of speed-ups using JCG and POAS solver on the Intel Delta for tracer flood.

Figure 8: Elapsed times for polymer flood on the CM-5 using JCG solver.

Figure 9: Parallel speed-ups for polymer flood on the CM-5 using JCG solver.