MATHS IN ENHANCED OIL RECOVERY

How to rev up an explicit time solver

Margot Gerritsen
Energy Resources Engineering
Stanford
Motivated by reservoir processes

Particular interest in Enhanced Oil Recovery processes

- Maintains pressure
- Swells oil phase
- Reduces oil viscosity

- $$$ Components transfer between phases

- Burns low percentage of oil
- Sets up steam flood, gas drive
- Upgrades oil in-situ

- $$$ kinetics
Effectiveness of injection process

1. *Global sweep*, mainly controlled by heterogeneity & gravity
   
   Low viscosity gas finds high perm flow paths
   Gravity segregates gas and oil phases

2. *Local displacement efficiency*, governed by phase behavior
   
   Strong nonlinear coupling (e.g. phase behavior)
   High sensitivity to *modeling* as well as *numerical* errors

Candidate for adaptivity, tailored computational methods:
tremendously fun area to be for a computational mathematician
Explicit methods desirable

- Upscaled models show large grid dependency
- Stiff reactions require specialized time-stepping
- Implicit methods, though industry standard, are out of the question
Need efficient explicit methods

Not one, but $O(100)$ realizations

*and*

High grid density required for accuracy because of strong sensitivities

Studies performed:
- Monte Carlo simulations
- History matching
- Sensitivities to input parameters
- Optimization (e.g. well placement)
**Governing system of nonlinear DAEs**

Multi-phase extension Darcy’s law

\[ u_j = -k\lambda_j (\nabla p + \rho_{mj} g), \quad \lambda_j = k_{rj}/\mu_j, \quad j = 1, \ldots, N_p, \]

Mass balance equations (transport equations)

\[ \phi \frac{\partial C_i}{\partial t} + \nabla \cdot U_i = q_i, \quad i = 1, \ldots, N_c. \]

\[ C_i = \sum_{j=1}^{N_p} x_i^j \rho_j S_j, \quad U_i = \sum_{j=1}^{N_p} x_i^j \rho_j u_j, \quad i = 1, \ldots, N_c, \]

Pressure equation (flow equation)

\[ c_t \frac{\partial p}{\partial t} + \sum_{i=1}^{N_c} \frac{\partial V_t}{\partial n_i} \nabla \cdot U_i = \sum_{i=1}^{N_c} \frac{\partial V_t}{\partial n_i} q_i. \]

Equation of state give mole fractions and densities
Asynchronous time stepping (FATI)

Standard approach:
All cells updated for same time step

Asynchronous:
Cells have their own update time
Ideal for localized time scales
Previous work

Localized time stepping

  Adaptive mesh refinement

  Iterative Mortar/domain decomposition methods

  Streamline methods

  Kirby and Dawson

Y. Omelchenko and H. Karimabadi (San Diego)

Using fluxes ensures conservation.

Target update time

Flux between cells

Current cell time
Move cell with smallest update time

minimum target cell
To update fluxes bring neighbors to same time level (partially)
But, are remaining time steps safe?
IF NOT, SYNCHRONIZE AND RESCHEDULE

Synchronization is recursive
**Simplified example: mathematics**

\[
\lambda_i^n = \frac{\Delta t_i^n}{\Delta x_i}
\]

\[
S_3^{n+1} = S_3^n - \lambda_3^n (F_3^{n+1} - F_2^n)
\]
\[
S_4^{n+1} = S_4^n - \lambda_4^n (F_4^n - F_3^n)
\]
\[
S_2^{n+1} = S_2^n - \lambda_2^n (F_2^n - F_1^n)
\]
\[
S_1^{n+2} = S_1^n - \lambda_1^n (F_1^n - F_0^n)
\]
\[
S_2^{n+2} = S_2^{n+1} - \lambda_2^{n+1} (F_2^{n+1} - F_1^n)
\]
\[
S_3^{n+2} = S_3^{n+1} - \lambda_3^{n+1} (F_3^{n+1} - F_2^n)
\]
\[
S_3^{n+3} = S_3^{n+2} - \lambda_3^{n+2} (F_3^{n+2} - F_2^n)
\]
\[
S_4^{n+3} = S_4^{n+1} - \lambda_4^{n+1} (F_4^{n+1} - F_3^n)
\]
\[
S_2^{n+3} = S_2^{n+2} - \lambda_2^{n+2} (F_2^{n+2} - F_1^n)
\]
Patterns can be irregular, but...

Cells with similar update times can be projected onto same time level

Efficiency improves when fast time scales are localized, e.g. around fronts/near wells
Euler-based scheme is inconsistent!

Linear advection
Single interface where time step jumps
Single point upwind flux

\[ S_{k}^{n+1} = S_{k}^{n} - \frac{\Delta t}{\Delta x} \left( G_{k+\frac{1}{2}} - G_{k-\frac{1}{2}} \right), \]

\[ G_{k-\frac{1}{2}} = S_{k-1}^{n}, \quad G_{k+\frac{1}{2}} = (1 - \beta) S_{k}^{n} + \beta S_{k-1}^{n} \]

\[ \beta = \frac{1}{p} \left( \frac{\lambda}{p + \lambda} \right) \left( 1 - \left( -\frac{\lambda}{p} \right)^{p-1} \right). \]
**First test – linear advection**

$p=2$

$p=4$

$\log N$

$\log \epsilon$

$L_2$ Error

Log $\epsilon$

Log $\epsilon$

Log $N$
Exaggerate effect inconsistency

![Graphs showing L2 and L∞ error vs. Log N for different synchronization types.]

- $p=2$
- $\log N$
- $\log e$
- $L^2$ Error
- $L^\infty$ Error
- $\log dt$
- $\log dt/p$
- Async

Log $N$ vs. Log $e$ for different synchronization types:
- Sync $dt$
- Sync $dt/p$
- Async
Two-phase, incompressible flow

\[
\begin{align*}
\text{Porosity} & = 27 \text{ (0.1 PVI), 13 (1.0 PVI)} \\
\text{Sync Updates} & = 2.5 \text{ (0.1 PVI), 1.0 (1.0 PVI)}
\end{align*}
\]
Red is asynchronous
Blue is synchronous
End time = 2/3 PVI

<table>
<thead>
<tr>
<th></th>
<th>CH4</th>
<th>CO2</th>
<th>C4</th>
<th>C10</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-Values</td>
<td>2.5</td>
<td>1.4</td>
<td>0.5</td>
<td>0.05</td>
</tr>
<tr>
<td>Cinj</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Cinit</td>
<td>0.1</td>
<td>0.1809</td>
<td>0.3766</td>
<td>0.3425</td>
</tr>
</tbody>
</table>
## Compositional results encouraging

Asynchronous tends to do about half the flash calculations of synchronous

<table>
<thead>
<tr>
<th>Grid Cells</th>
<th>Flash Calculations</th>
<th>Flux Calculations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number</td>
<td>Time</td>
</tr>
<tr>
<td><strong>100</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>19261</td>
<td>2.8</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>8367</td>
<td>2.1</td>
</tr>
<tr>
<td><strong>500</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>466281</td>
<td>58.9</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>149657</td>
<td>33.7</td>
</tr>
</tbody>
</table>
Recursion is largely localized

Turning off recursions seems to have little effect on the solution. However, stability analysis still must be performed (ongoing work).
An initial 2D Algorithm

Miscible and immiscible two phase flow
Incompressible, no gravity or capillarity
Compare sequential algorithms

Synchronous vs. Asynchronous

Polygonal grids with L-method, upstream weighting
dt chosen so that CFL satisfied for 95% of cells
2D Results

<table>
<thead>
<tr>
<th></th>
<th>Sync updates</th>
<th>Sync flux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immiscible</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>Miscible</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

Saturation 0.4 PVI

Immiscible Miscible
FATI requires a priority queue

At each iteration, update cell with nearest scheduled final time
(Discrete Event Simulation)

Cells must be kept ordered by scheduled final time time

But must also modify neighbours, which may be anywhere

The priority queue drives FATI and is classically implemented as a *minheap*

Bottleneck? Parallelization?
Priority Queue

Minheap seems expensive

Heap operations are \( \sim 50\% \) runtime

**But**

1D/2D runtimes are only 20-35\% of synchronous equivalent

Cell calculations are currently trivial

Heap cost is \( \log(\text{number of cells}) \]

No bottleneck with large domains / more realistic problems

Parallelization (needed) is a different story!
The devil is in the implementation!

So optimistic…..

Multi-core Era: A new paradigm in computing

Massively Parallel Era
- USA, Japan, Europe

Vector Era
- USA, Japan

IAA (Sandia/ORNL)
Silicon valley moves to CMPs

Flops are free.  
In most cases we can now compute on the data as fast as we can move them.  
Doug Miles, Portland Group

Power/cooling limits constrain clock speed

Industry moves to chip-level multiprocessors:
• Two or more cores in single package
• Cores may share cache (e.g. L2)
• Cores share interconnect to rest system

Designed for specific workloads (mail, encryptions, database)
1. *Commercial simulators*

   Eclipse - Schlumberger, GEM/STARS - CMG

2. *Petroleum companies in house codes*

3. *Academic home brews*

Commercial codes drivers for first vector machines (eg Cray)
Commercial codes are legacy codes, parallelized, Fortran core code

“Next generation” software under development (90s technology)
Implemented on large distributed clusters (MPI based mostly)

Academic homebrew heavily rely on MATLAB, C++
Some parallelized (usually MPI, some openMP)
IAA Mission and Strategy

IAA is being proposed as the medium through which architectures and applications can be co-designed in order to create synergy in their respective evolutions.

- Focused R&D on key impediments to high performance in partnership with industry and academia
- Foster the integrated co-design of architectures and algorithms to enable more efficient and timely solutions to mission critical problems
- Partner with other agencies (e.g., DARPA, NSA ...) to leverage our R&D and broaden our impact
- Impact vendor roadmaps by committing National Lab staff and funding the Non-Recurring Engineering (NRE) costs of promising technology development and thus lower risks associated with its adoption
- Train future generations of computer engineers, computer scientists, and computational scientists, thus enhancing American competitiveness
- Deploy prototypes to prove the technologies that allow application developers to explore these architectures and to foster greater algorithmic richness
What next?

• Analyze the nonlinear algorithm
  – Monotonicity & stability
• Understand difference between CFL and recursion
  – What happens if recursion is limited?
  – How “safe” do we need to be?
  – Can we break one (or both) methods?
• More complex physics & flux calculation
  – Counter current flow
  – Higher-order methods
  – More phase behavior
• Parallelization
Parallelization strategy FATI?

Queue-driven boss-worker (or master-slave)

Fixed sub-domain size (at first)
  Statically determined extents
  Dynamically determined extents

Can experiment with optimum work-unit size to suit cache/comms
  Bring sub-domain up to end-time of smallest member cell cheaply?

Workers have their
  own
  minheap whose root is heaped by the boss's heap, distributing heap cost