Parallelization for an RBC Model

The model


Households

Households choose consumption $c_t$ and capital $k_t$ to maximize

$$ E_0 \sum_{t=0}^{\infty} \beta c_{t+1} \eta k_t $$

subject to a budget constraint

$$ c_t + it = w_t + r k_t $$

law of motion for capital:

$$ k_{t+1} = (1 - \delta) k_t + it $$

Production

Representative firm with technology

$$ y_t = z_t k_t $$

Technological shock:

$$ \log z_t = \log z_{t-1} + \epsilon_t \sim N(0, \sigma^2) $$

Recursive formulation:

Select next period capital to maximize the value function:

$$ V(k_{t+1}, z_{t+1}) = \max_k u(c(k_t, k_{t+1})) + \beta E[V(k_{t+1}, z_{t+1})|z] $$

Where

$$ c(k_t, k_{t+1}) = z k_t + (1 - \delta) k_t $$
Notation: define continuation value $\psi(k_{t+1},z)$ as the present expected value of investing $k_t$ for the next period, given that we are today in state $z$:

$$\psi(k_{t+1},z) = \beta E\{V(k_{t+1},z)\mid z\}$$

the optimization problem then becomes

$$V(k,z) = \max_{k_t} u(c(k_t,k_{t+1}))+\psi(k_{t+1},z)$$

...if we knew the continuation value, we could solve this for $V$!

### Numerical Solution

#### Value function iteration: theory

- Guess the value function at $t=0$ and proceed backwards
  - i.e. to time $t=-1, -2, -3, \ldots$
  - if we go far enough, initial guess does not matter anymore!
- Define $V_0(k,z) = 0$
- For all $t=1, -2, \ldots$ :
  - Compute $\psi_{t-1}(k_{t+1},z) = \beta E_{t-1}\{V_t(k_{t+1},z)\mid z\}$
    - This is the present expected value of investing $k_t$, given that today we are in state $z$
  - Compute $V_{t-1}(k,z) = \max_{k_t} u(c(k_t,k_{t+1}))+\psi(k_{t+1},z)$
  - Repeat until $\max_{k_t,z}\|V_t(k,z) - V_{t-1}(k,z)\| < \epsilon$
- Once we have the value function, we have everything!

#### Value function iteration: implementation

- Problem is discretized and solved on a grid of $k$ and $z$
- Grid search each period to find the maximum
- Advantages:
  - Simple to implement
  - An approximate model, solved exactly
    - as opposed to an exact model, solved approximately
  - Does not rely on interpolation
  - Does not rely on convex solvers
    - Robust to local maxima
  - Robust to kinks and jumps (e.g. liquidity constraints)
- Disadvantages: slow!
  - Many ways to improve the speed by a lot
  - But let us try parallelization first...
- This is an illustrative example, **not** the optimal method!
Testing System

- Dual Intel Xeon X5670 (2x6 = 12c at 2.8GHz), 24G RAM
- NVidia GFX480 GPU (480 CUDA cores at 700MHz, 1.5G RAM)
- Fedora 13, Intel compiler suite 11.1, GCC 4.4.5

Languages: Matlab and Fortran

Matlab

- Easy to use
- Great documentation, built-in graphics
- Extensive toolbox library
- Requires Matlab Parallel Toolbox for parallelization
  - Available in SSCC, Quest, Kellogg desktop installations
  - Limited to max. 12 cores (as of R2011b)
    - can do more, but with a (very) expensive server
  - Invoke matlabbpool open to enable parallel processing
- Some operations (matrix multiplication, optimization routines) are partially parallelized
- Also has (very limited) GPU functionality

Fortran basics

- Fortran is one of the oldest programming languages
- Many existing codes, libraries, etc.
- Substantial additions and updates over the years (ongoing)
  - Fortran 66, 77, 90, 95, 2003, 2008, ...
- Syntax similar to Matlab
  - Especially Fortran 90 and later
  - Built-in array types, array math, and vector indexing
  - Array indexes can start anywhere: A(-5:2, 13:18)
  - First index changes fastest (i.e. storage by column)
- Popular compilers: Intel (icc), gfortran, PGI (pgf90)
- Can be combined with Matlab or other languages
  - save a file in Matlab, process in Fortran, read computed results back into Matlab for plotting, etc.

Fortran vs. Matlab: matrix algebra

Here, A, B,... are matrices; a,b,... are vectors.
### Matlab vs. Fortran 95: Control Statements

<table>
<thead>
<tr>
<th>Matlab</th>
<th>Fortran 95</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for i=1:10; ...; end</code></td>
<td><code>do i=1,10 .... end do</code></td>
</tr>
<tr>
<td><code>if i==j; ...; else ... ; end</code></td>
<td><code>if(i==j) then... ; else... ; end if</code></td>
</tr>
<tr>
<td><code>x =f(y)</code></td>
<td><code>x =f(y)</code></td>
</tr>
<tr>
<td><code>function x =f(y)</code></td>
<td><code>real function f(y)</code></td>
</tr>
<tr>
<td></td>
<td><code>real :: y</code></td>
</tr>
<tr>
<td><code>[x,y] =f(z)</code></td>
<td><code>call f(x,y,z)</code></td>
</tr>
<tr>
<td><code>function [x,y] =f(z)</code></td>
<td><code>subroutine f(x,y,z)</code></td>
</tr>
<tr>
<td></td>
<td><code>real :: x,y,z</code></td>
</tr>
<tr>
<td><code>global x y z</code></td>
<td><code>common x,y,z</code></td>
</tr>
</tbody>
</table>

### Sequential Code

\[ A = 2 + 3 \times B \]
\[ A = 2 + 3 \times B \]
\[ A(:, 2:end) \]
\[ A(:, 2:) \]
Many options

- Languages, compilers and libraries matter
- Matlab: great documentation, can run in parallel
  - OK for "vectorized" programs, slow in loops
- Fortran 90: syntax similar to Matlab
- C: the classic
- R: itself slow, but has many packages, easy to combine with C and

Floating-point precision

- Most common format is double precision (8 bytes)
  - Matlab format is double precision
- For some models, single precision (4 bytes) is enough
  - C: "float" vs. "double"
  - Fortran: "real" vs. "double precision"
    - Force double: -r8 (ifort, PGI), -fdefault-real-8 (gfortran)
- The gain is higher on GPUs
  - especially older models
- Fortran also has "quad precision" (16 bytes, slow)
- Make sure you choose precision that is sufficient for your problem!

Matlab implementation

% Initialize the parameters
beta = 0.984; % Discount rate
et = 2; % Risk aversion parameter
alpha = 0.35; % Technology parameter
delta = 0.01; % Depreciation rate
rho = 0.95; % Tech. shock persistence
sigma = 0.005; % Tech. shock st. dev.

Discretizing productivity

- Gaussian random variable is approximated by a 4-state Markov chain

% Grid for productivity z
nz = 4; % Grid size
zmin=-0.0480384;
zmax=0.0480384;
% [1 x 4] grid of values for z
\[
\text{zgrid} = \exp(\text{zmin}:((\text{zmax}-\text{zmin})/(\text{nz}-1)):\text{zmax})
\]
% [4 x 4] Markov transition matrix of z
\[
\text{tran}_z = \begin{bmatrix}
0.996757 & 0.00324265 & 0 & 0 \\
0.000385933 & 0.998441 & 0.00117336 & 0 \\
0 & 0.00117336 & 0.998441 & 0.000385933 \\
0 & 0 & 0.00324265 & 0.996757
\end{bmatrix};
\]

### Discretizing capital

- Grid for capital is the primary determinant of accuracy
- Baseline: \( nk = 4800 \) points
  - Somewhat of an overkill in this case
  - In slightly more complex problems, grids can easily get much larger!
  - With \( N \) states we have an \( N \)-dimensional grid
  - Why 4800? It is nice to have the grid size divisible by the number of cores in the CPU (12) or GPU (480).

\[
\text{\% Grid for capital } k
\]
\[
k = 4800; \quad \% \text{Number of points for capital}
k_{\text{min}} = 0.95 \times (1/(\alpha \times \text{zgrid}(1))) \times ((1/\beta)-1+\delta)^{(1/(\alpha-1))};
k_{\text{max}} = 1.05 \times (1/(\alpha \times \text{zgrid}(\text{end}))) \times ((1/\beta)-1+\delta)^{(1/(\alpha-1))};
k_{\text{step}} = (k_{\text{max}} - k_{\text{min}})/(nk-1);
\]
\[
\% [1 \times 4800] \text{ grid of possible values of } k
\]
\[
k_{\text{grid}} = k_{\text{min}}:k_{\text{step}}:k_{\text{max}};
\]

### Initial values

\[
\% \text{ Compute the solution: Sequential}
\]
\[
tic \quad % <------- Start the timer
v = zeros(nk,nz); \quad % \text{Value Function}
v_0 = zeros(nk,nz); \quad % v at the previous iteration
v_0 = zeros(nk,nz); \quad % \text{continuation value}
c_0 = zeros(nk,nz); \quad % \text{total wealth}
c = zeros(nk,1); \quad % c(k'), given (k,z)
\]
\[
% \text{Compute initial wealth } c_0(k,z)
\text{for } iz=1:nz;
\quad c_0(:,iz) = \text{zgrid}(iz) \times k_{\text{grid}}.^{\alpha} + (1-\delta) \times k_{\text{grid}};
\text{end}
\]
\[
% \text{Define the utility function}
u_\text{f} = @(c) c.^{(1-\text{eta})}/(1-\text{eta});
\]
Solve the problem

tol = 1e-8;                       % Tolerance for V
cnt = 1;                          % Iteration counter
while(1)
    for iz=1:nz;                  % For all values of z
        for ik = 1:nk;            % For all values of k
            c = c0(ik,iz)-kgrid;  % Compute c(k')
            ind = c>0;            % Check if c(k') > 0
            % Grid search to find v(k,z):
            v(ik,iz) = max(uf(c(ind))+ev(ind,iz)');
        end
    end
    ev = beta*v*tran_z';          % Compute the expectation
% Check convergence:
dif = max(abs(v(:)-v0(:)));
v0 = v;                       % Save the value function
if(~mod(cnt,100)) fprintf('%d : %f
',cnt,dif); end;
cnt = cnt+1;
if(dif<tol) break; end;
end
toc  % <----- Stop the timer

Results

• Time: 3718.57 seconds
• What if we need to compute this for different values of parameters?

Fortran implementation

Declarations

Need to declare variables first:

program rbc
    use omp_lib     ! For timing
    implicit none

    real, parameter :: beta = 0.984, eta = 2, alpha = 0.35, delta = 0.01, &
        rho = 0.95, sigma = 0.005, zmin=-0.0480384, zmax=0.0480384;
    integer, parameter :: nz = 4, nk=4800;
    real :: zgrid(nz), kgrid(nk), t_tran_z(nz,nz), tran_z(nz,nz);
real :: kmax, kmin, tol, dif, c(nk), r(nk), w(nk);
real, dimension(nz,nk) :: v=0., v0=0., ev=0., c0=0.;
integer :: i, iz, ik, cnt;
logical :: ind(nk);
real(kind=8) :: start, finish   ! For timing
real :: tmpmax, c1

Main loop

<table>
<thead>
<tr>
<th>Matlab</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(1)</td>
<td>do while(dif&gt;tol)</td>
</tr>
<tr>
<td>for iz=1:nz;</td>
<td>do iz=1,nz;</td>
</tr>
<tr>
<td>for ik = 1:nk;</td>
<td>do ik = 1,nk;</td>
</tr>
<tr>
<td>c = c0(ik,iz)-kgrid;</td>
<td>c = c0(ik,iz)-kgrid;</td>
</tr>
<tr>
<td>ind = c&gt;0;</td>
<td>ind = c&gt;0;</td>
</tr>
<tr>
<td>v(iz,iz) = max(uf(c(ind))+ev(ind,iz)');</td>
<td>v(iz,iz) = maxval(pack(c,ind)**(1-eta)/(1-eta)+pack(ev(:,iz), ind));</td>
</tr>
<tr>
<td>end</td>
<td>end do</td>
</tr>
<tr>
<td>end</td>
<td>end do</td>
</tr>
<tr>
<td>ev = beta<em>v</em>tranz';</td>
<td>ev = beta*matmul(v,tranz)</td>
</tr>
<tr>
<td>dif = max(abs(v()-v0()));</td>
<td>dif = maxval(abs(v-v0))</td>
</tr>
<tr>
<td>v0 = v;</td>
<td>v0 = v</td>
</tr>
<tr>
<td>if(~mod(cnt,100)) fprintf('%d : %f\n;', n',cnt,dif); end;</td>
<td>if(mod(cnt,100)==0) write(<em>,</em> cnt, ',', dif</td>
</tr>
<tr>
<td>cnt = cnt+1;</td>
<td>cnt = cnt+1</td>
</tr>
<tr>
<td>if(dif&lt;tol) break; end;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>end do</td>
</tr>
</tbody>
</table>
Loops are faster in Fortran!

This "vectorized" piece is valid Fortran code:

```fortran
  c = c0(ik,iz)-kgrid;
  ind = c>0;
  v(ik,iz) = maxval(pack(c,ind)**(1-eta)/(1-eta)+pack(ev(:,iz),
          ind));

  An explicit loop over k can avoid computation for k>c0:

  tmpmax = -huge(0.)
  do i = 1,nk
    c1 = c0(ik,iz) - kgrid(i)
    if(c1<0) exit
    c1 = c1**(1-eta)/(1-eta)+ev(i,iz)
    if(tmpmax<c1) tmpmax = c1
  end do
  v(ik,iz) = tmpmax

  This would be (much) slower in Matlab, but faster in Fortran.
```

Timings

There are many Fortran and C compilers...

<table>
<thead>
<tr>
<th>Flag</th>
<th>ifort/icc</th>
<th>pgf95/pgcc</th>
<th>gfortran/gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default double (Fort. only)</td>
<td>-r8</td>
<td>-r8</td>
<td>-fdefault-real-8</td>
</tr>
<tr>
<td>Auto-optimize for speed</td>
<td>-fast</td>
<td>-fast</td>
<td></td>
</tr>
<tr>
<td>Extra optimizations</td>
<td>-O3</td>
<td>-O3</td>
<td>-O3</td>
</tr>
<tr>
<td>Optimize for recent Intel</td>
<td></td>
<td></td>
<td>-mtune=core2</td>
</tr>
<tr>
<td>Optimize for recent AMD</td>
<td></td>
<td></td>
<td>-mtune=amdfam10</td>
</tr>
<tr>
<td>Use &quot;fast math&quot;</td>
<td></td>
<td></td>
<td>-ffast-math</td>
</tr>
<tr>
<td>Link math library (C only)</td>
<td></td>
<td></td>
<td>-ml</td>
</tr>
<tr>
<td>Use OpenMP</td>
<td>-openmp</td>
<td>-mp</td>
<td>-fopenmp</td>
</tr>
</tbody>
</table>

- Compiler string:
  - ifort -r8 -fast -O3 rbc.f90
- Time: 707.87 sec (5.25x Matlab)
Parallelized code

Matlab implementation

Matlab parallel computing and parfor

- Matlab parallel toolbox:
  - Allows to run code in parallel across multiple CPUs/cores
  - skew4, Quest, SSCC, Kellogg desktop licenses
  - Without Matlab server, limited to 8 "local" threads
- Easiest way to parallelize: use parfor in place of for
  - The index of the parfor loop cannot be combined with any other!
  - Creates some communication overhead
  - Requires additional programming
- parfor is just for when running sequentially
  - But overhead is still there...

Matlab: parallelizing code

Parallelized code requires some adjustments in Matlab:

```matlab
parfor ik=1:nk; % Put the longer loop first
  vtmp = zeros(nz,1); % Temporary
  cc = c0(ik,:); % Temporary
  c = zeros(nk,1); % Fill with zeroes
  ind = zeros(nk,1); % Fill with zeroes
  for iz = 1:nz;
    c = cc(iz)-kgrid;
    ind = c>0;
    vtmp(iz) = max(uf(c(ind))+ev(ind,iz)');
  end
  v(ik,:) = vtmp; % Assign to v
end
```

- Time: 478.94 sec (7.8x sequential Matlab, 1.5x seq. Fortran)
- On a multi-core machine, faster than (sequential) compiled code!
Fortran implementation

Fortran and OpenMP

- An industry standard for parallelization in C and Fortran
- Implemented as comments with compiler directives
- Supported by all major compilers
- Easy to program
- Allows sharing variables between threads
- Limited to shared-memory computers (e.g. a single node)

Fortran: parallelizing code (OpenMP)

The only difference is the two comment lines!

```
!$omp parallel do default(shared) private(ik,iz,i,tmpmax,c1)
do ik=1,nk;
do iz = 1,nz;
tmpmax = -huge(0.)
do i = 1,nk
   c1  = c0(ik,iz) - kgrid(i)
   if(c1<0) exit
   c1 = c1**(1-eta)/(1-eta)+ev(i,iz)
   if(tmpmax<c1) tmpmax = c1
   end do
   v(ik,iz) = tmpmax
end do
!$omp end parallel do
```

We need to tell the compiler that we want to parallelize with OpenMP:

- ifort -r8 -fast -O3 -openmp rbc.f90
- gfortran -fdefault-real-8 -O3 -openmp -ffast-math rbc.f90
- pgf95 -r8 -fast -O3 -mp rbc.f90

Timings

- Sequential Matlab: 3718.6 sec
- Sequential Fortran: 707.9 sec
- Parallel Matlab: 478.94 sec
- Parallel Fortran: 66.22 sec
GPU computation

PGI directives

We'll use PGI compiler directives. The PGI compilers are available on SSCC. Note that we only had to add four extra lines!!

!$acc region
!$acc do parallel
do ik = 1,nk;
    !$acc do parallel
    do iz=1,nz;
        tmpmax = -huge(0.)
        do i = 1,nk
            cl = c0(ik,iz) - kgrid(i)
            if(cl<0) exit
            cl = cl**(1-eta)/(1-eta)+ev(i,iz)
            if(tmpmax<cl) tmpmax = cl
        end do
        v(ik,iz) = tmpmax
    end do
end do
!$acc end region