High Performance Computing

Autumn, 2018

Lecture 14

Imperial College
London

Prasun Ray

20 November 2018

OpenMP

What directives/routines have been covered?

OpenMP

Directives:
!$OMP parallel
!$OMP parallel do
!$OMP do
!$OMP critical
!$OMP single
!$OMP sections

Routines:
omp_get_num_threads
omp_set_num_threads
omp_get_thread_num

reduction, private, firstprivate

Notes

Homeworks:

- HW1: Will respond to e-mail queries on marking by end of term (probably the last week of term)
  - Scores may go up or down

- HW2: Solutions should be posted end of this week, marks end of next week

- HW3: Posted today around 8pm
  - Due next **Thursday** (so you have extra time for final project)
  - Will require f2py + fortran + openmp (Try today's lecture code!)
  - Will need ffmpeg to save animations
    - Installation instructions will be posted on course webpage
    - Demo during this week's lab on Huxley 410 VM

XUbuntu VMs:

- Installed directly on Huxley 408 and 410 machines
- Available via software hub in MLC

Today

A little more on OpenMP: synchronization, thread-safe subroutines, nested for loops

Programming example: from PDE → algorithm → serial code → parallel code

Parallel loops: nested loops

Must always be sure loop(s) can be parallelized

Example:

```
!$OMP parallel do private(j1)
do i1 = 1,M
    do j1 = 2,N
        x(i1,j1) = x(i1,j1-1)
    end do
end do
!$OMP end parallel do
```

Correct

- **Solution: swap inner and outer loops**
- **Now, computation of x is "safe."**
  - **The "i1 loop" is parallelized, and calculations of x do not depend on the order in which i1 is iterated.**

Parallel loops: nested loops

Must always be sure loop(s) can be parallelized

Example:

```
!$OMP parallel do collapse(2)
do i1 = 1,M
    do j1 = 2,N
        x(i1,j1) = sin(y(i1,j1))
    end do
end do
!$OMP end parallel do
```

- **Nested loops can be “collapsed”**
  - **If both loops are parallelizable**
  - **And there is no code “between” the loops**

Synchronization

- **Some threads may be given more work than others**
- **One thread may complete its tasks quickly and move very far ahead of the other threads**
- ***Barriers* keep the threads synchronized:**

```
!$OMP parallel

!Some code

!$OMP barrier

!$OMP end parallel
```

- **Threads will not continue past the barrier until all threads reach the barrier**

Synchronization

- **Some threads may be given more work than others**
- **One thread may complete its tasks quickly and move very far ahead of the other threads**
- ***Barriers* keep the threads synchronized:**

```
!$OMP parallel

!Some code

!$OMP barrier

!$OMP end parallel
```

- **Threads will not continue past the barrier until all threads reach the barrier**
- **There are *implicit* barriers at end of** !$OMP do **and** !$OMP single **blocks**

Thread-safe routines

- **What happens when you call sub-program from within parallel region?**
- **Each thread will call it's own "copy" of sub-program**
  - **All "local" variables declared within sub-program are private to thread**

```
!$OMP parallel
call sub1(in1,in2,out1,out2)
!$OMP end parallel
!---------------------------------
subroutine sub1(in1,in2,out1,out2)
    use mod1
    implicit none
    real(kind=8) intent(in) :: in1,in2
    real(kind=8) intent(out) :: out1,out2
    real(kind=8) :: local1

    !should not modify mod1 variables
    !out1,out2 should (usually) be
    !private in the calling parallel region

end subroutine sub1
```

Basic questions:

1. **Does code give same answer independent of the total number of threads?**

2. **Is it independent of the *order* in which threads call the subroutine**

If yes, the subroutine is *thread-safe*

Should not include OMP directives in subroutine called from within parallel region

Vectorizing code

In general (Python, Fortran, Matlab,...), avoid for loops and *vectorize* calculations involving arrays.

Example:

```
In [27]: x=np.linspace(0,1,101)

In [28]: f = np.empty_like(x)

In [29]: for i in range(size(x)):
   ....:         f[i] = cos(x[i])**2
   ....:

In [30]: f = cos(x)**2
```

Vectorized version of loop

Vectorizing code

In general (Python, Fortran, Matlab,…), avoid for loops and *vectorize* calculations involving arrays.

Example:

```
In [27]: x=np.linspace(0,1,101)

In [28]: f = np.empty_like(x)

In [29]: for i in range(size(x)):
    ....:         f[i] = cos(x[i])**2
    ....:

In [30]: f = cos(x)**2
```

Vectorized version of loop

- **Vectorized code will usually be faster, sometimes *much faster* in interpreted languages**
- **Exception: parallelizing Fortran code with OpenMp:**
  vectorized code → loops → parallel loops

Programming example

Task: Compute temperature distribution in a room

Programming example

Task: Compute temperature distribution in a room

Governing equation: Heat equation (diffusion equation):

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T + S(\mathbf{x}, t)$$

$$T(\mathbf{x}, t = 0) = f(\mathbf{x})$$ **Initial condition**

Here, S **is a *heat source.* Boundary conditions should also be specified as appropriate.**

Problem: given the source, initial condition, and boundary conditions, solve for the temperature distribution, T(**x**,t)

Programming example

Today: 1-D problem

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + S(x,t)$$

$$T(x, t=0) = f(x)$$ **Initial condition**

$$T(x=0,t) = a(t), \;\; T(x=1,t) = b(t)$$ **Boundary conditions**

$$0 \leq x \leq 1$$

Programming example

Today: 1-D problem

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + S(x,t)$$

$$T(x, t=0) = f(x)$$ **Initial condition**

$$T(x=0,t) = a(t), \;\; T(x=1,t) = b(t)$$ **Boundary conditions**

$$0 \leq x \leq 1$$

First consider steady problem, e.g., S = S(x)**,** a **and** b **are constants:**

$$\frac{\partial^2 T}{\partial x^2} + S(x,t) = 0$$ **Poisson equation**

Programming example

First consider steady problem, e.g., S = S(x), a **and** b **are constants:**

$$\frac{\partial^2 T}{\partial x^2} + S(x) = 0$$

Notes:

1. **This is an extremely simple problem, easy to write down the analytical solution**
2. **No need to use compiled language**
3. **Certainly no need to parallelize**
4. **But what about two-dimensional or three-dimensional problems?**
    - **Then, the picture changes considerably!**
5. **We are just considering the 1-D problem for illustrative purposes**

Programming example

First consider steady problem, e.g., S = S(x)**,** a **and** b **are constants:**

$$\frac{\partial^2 T}{\partial x^2} + S(x) = 0$$

Poisson equation

Numerical method:

1. Discretize the derivative:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$

2nd-order, centered scheme

$$x_i = i * \Delta x,\ i = 1, 2, ..., N$$

$$(N+1) * \Delta x = 1$$

Programming example

First consider steady problem, e.g., S = S(x)**,** a **and** b **are constants:**

$$\frac{\partial^2 T}{\partial x^2} + S(x) = 0$$

Poisson equation

Numerical method:

1. **Discretize the derivative:**

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$

2nd-order, centered scheme

$$x_i = i * \Delta x,\ i = 0, 1, 2, ..., N+1$$

$$(N+1) * \Delta x = 1$$

With boundary conditions: $T_0 = T_a,\ T_N = T_b$

Programming example

Equation for T_i: $$\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} = -S_i$$

In matrix form: AT = b

$$A = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & \dots & 0 & 0 & 0 & 1 & -2 \end{bmatrix}, b = \Delta x^2 \begin{bmatrix} -T_a/\Delta x^2 - S_1 \\ -S_2 \\ \vdots \\ -S_i \\ \vdots \\ -S_{N-1} \\ -T_b \Delta x^2 - S_N \end{bmatrix}$$

- **In 1-D, this is just a tridiagonal system of equations**
- **Easy to solve directly (with, say, *DGTSV*)**

Programming example

- **In two or three dimensions, A loses it simple banded structure**
- **Then, direct solution becomes very expensive for large** N
- ***Iterative* methods are a popular alternative**

Programming example

- In two or three dimensions, A loses it simple banded structure
- Then, direct solution becomes very expensive for large N
- *Iterative* methods are a popular alternative
- Basic idea: rewrite $Ax=b$ as $A_1x = A_2x + b$
- *Choose* A_1 so that it is easy to invert, then solve iterative system:
- $A_1x^{k+1} = A_2x^k + b$
    - Requires guess, x^0

Jacobi iteration

- **Basic idea: rewrite Ax=b as $A_1x = A_2x + b$**
- ***Choose* A_1 so that it is easy to invert, then solve iterative system:**
- **$A_1x^{k+1} = A_2x^k + b$**
  - **Requires guess, x^0**
- ***Jacobi iteration*: Choose A_1 to be diagonal matrix (main diagonal of A):**

$$\frac{T_{i+1}^{k-1} - 2T_i^k + T_{i-1}^{k-1}}{\Delta x^2} = -S_i$$

$$T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)$$

Jacobi iteration

- Basic idea: rewrite Ax=b as $A_1x = A_2x + b$
- *Choose* A_1 so that it is easy to invert, then solve iterative system:
- $A_1x^{k+1} = A_2x^k + b$
  - Requires guess, x^0
- *Jacobi iteration*: Choose A_1 to be diagonal matrix (main diagonal of A):

$$\frac{T_{i+1}^{k-1} - 2T_i^k + T_{i-1}^{k-1}}{\Delta x^2} = -S_i$$

$$\boxed{T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)}$$

Main algorithm, easy to code!

Jacobi iteration in Fortran

- **Plan:**
    - **Set parameters:** a, b, n, tol
    - **Construct grid** x_i
    - **Construct source function,** S(x)**, initialize** T=T(x,t=0)
    - **Iterate using formula below**
        - **Each iteration check if** |Tk-Tk-1| **<** tol

$$\boxed{T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)}$$

Jacobi iteration in Fortran

- **One Fortran trick: set variables to be** dimension(0:N+1)
  - x(0)=0, x(N+1)=1, T(0)=a, T(N+1)=b
  - **Then, easy to compute** T_1 **using:**

$$\boxed{T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)}$$

Jacobi iteration in Fortran

- **One Fortran trick: set variables to be** dimension(0:N+1)
  - x(0)=0, x(N+1)=1, T(0)=a, T(N+1)=b
  - **Then, easy to compute** T_1 **using:**

$$\boxed{T_i^k = \frac{\Delta x^2}{2} S_i + \frac{1}{2}\left(T_{i+1}^{k-1} + T_{i-1}^{k-1}\right)}$$

Core part of code (see *jacobi1s.f90)*:

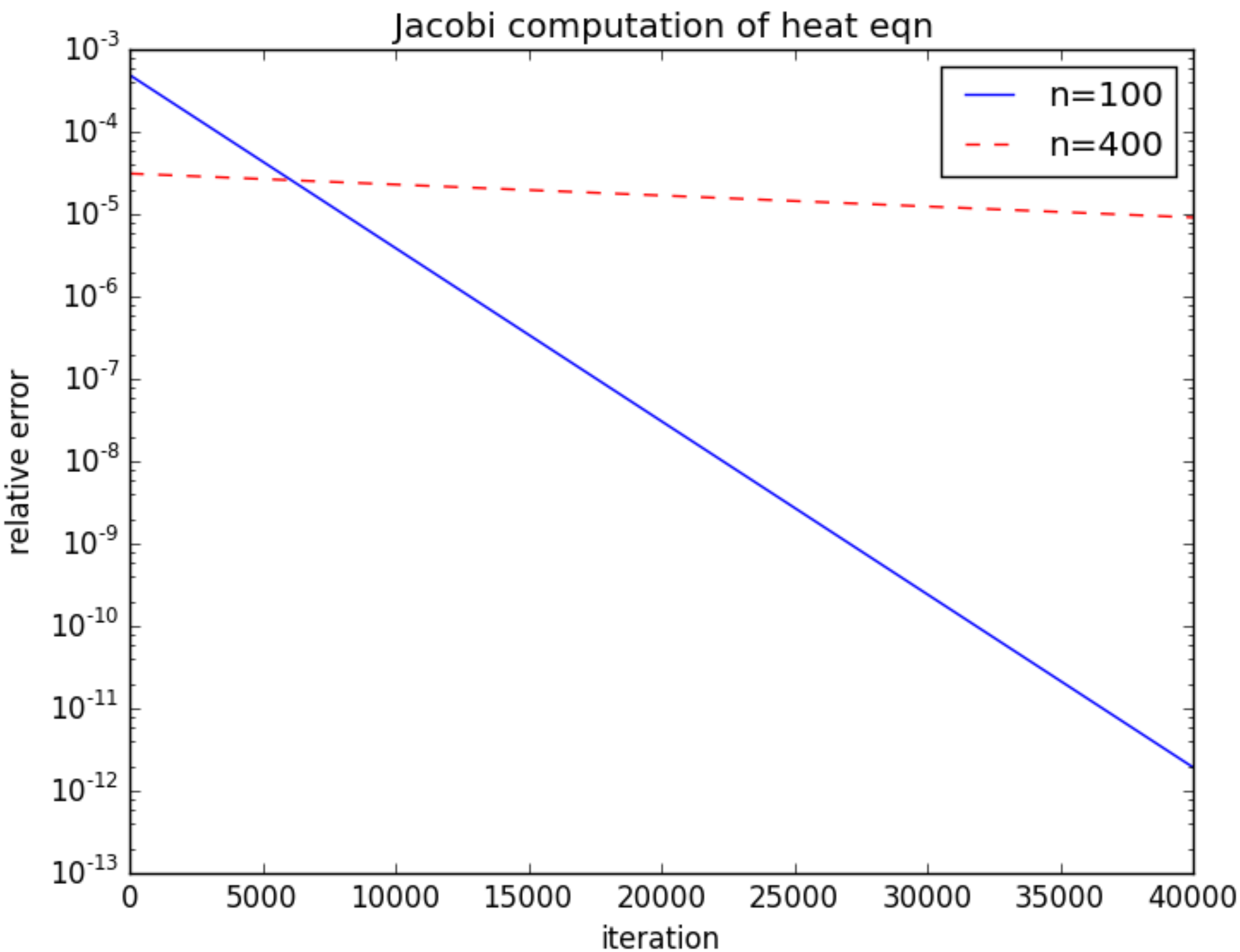
```
do k1=1,kmax
   Tnew(1:n) = S(1:n)*dx2f + 0.5d0*(T(0:n-1) + T(2:n+1)) !Jacobi
   deltaT(k1) = maxval(abs(Tnew(1:n)-T(1:n))) !compute relative error
   T(1:n)=Tnew(1:n)    !update variable
   if (deltaT(k1)<tol) exit !check convergence criterion
end do
```

Jacobi results

1. Does the solution converge?

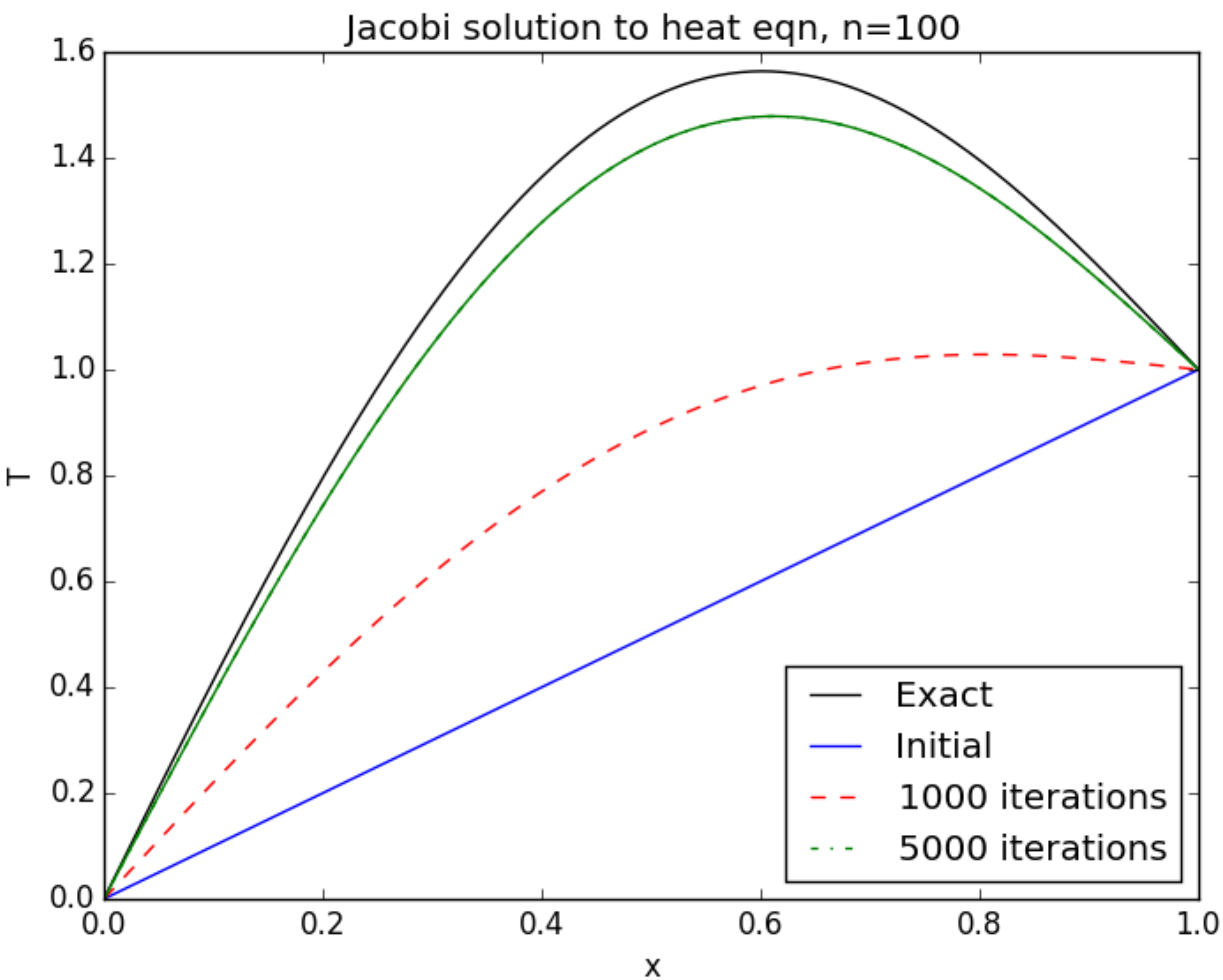
Jacobi results

1. Does the solution converge? Yes, but very slowly for large n



Jacobi results

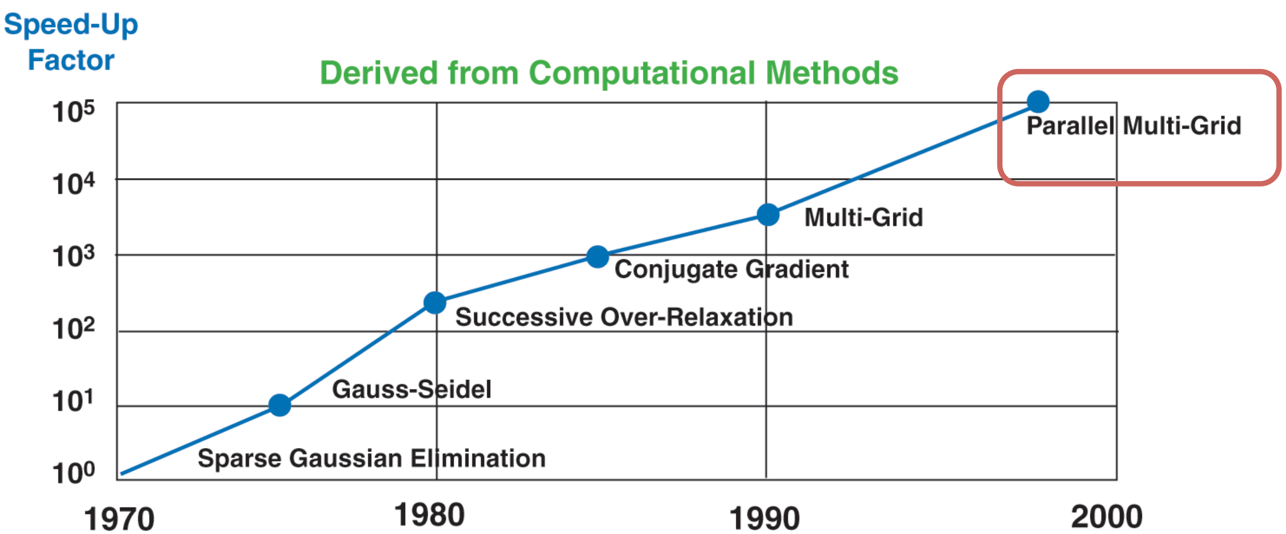
1. **Does it converge to the correct solution? Yes, can check that error ~ Δx^2**

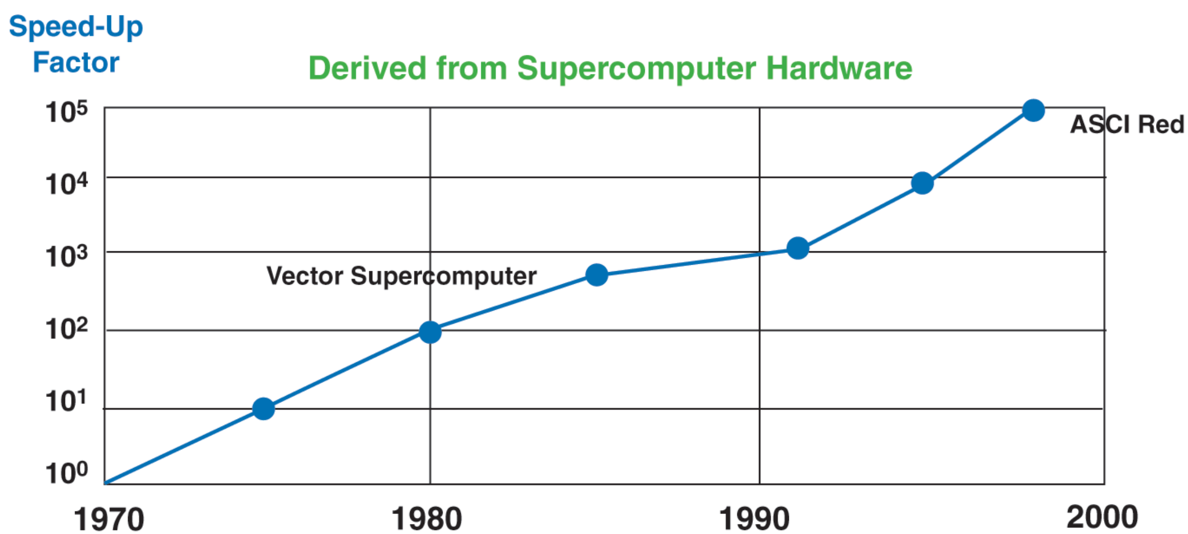


Jacobi results

- Jacobi is simplest, but *most inefficient* iterative solver
- Good illustration of basic ideas
- Better methods: Gauss-Seidel, SOR, conjugate gradient, multigrid

Algorithms and hardware





SIAM Rev (2001)

Parallel Jacobi

Let's now parallelize the solver with OpenMP

- **Look for loops that can be parallelized**
- **Look for vectorized operations that can be converted to loops that can be parallelized**

Serial:

```
Tnew(1:n) = S(1:n)*dx2f + 0.5d0*(T(0:n-1) + T(2:n+1)) !Jacobi

deltaT(k1) = maxval(abs(Tnew(1:n)-T(1:n))) !compute relative error
```

Parallel Jacobi

Let's now parallelize the solver with OpenMP

- **Look for loops that can be parallelized**
- **Look for vectorized operations that can be converted to loops that can be parallelized**

Parallel:

```
 dmax=0.d0
!$omp parallel do reduction(max:dmax)
do i1=1,n
    Tnew(i1) = S(i1)*dx2f + 0.5d0*(T(i1-1) + T(i1+1))
    dmax = max(dmax,abs(Tnew(i1)-T(i1)))
end do
!$omp end parallel do
deltaT(k1) = dmax
```

Parallel Jacobi

Let's now parallelize the solver with OpenMP

- **Look for loops that can be parallelized**
- **Look for vectorized operations that can be converted to loops that can be parallelized**

Serial:

```
do i1=0,n+1
    x(i1) = i1*dx
end do
!-------------------

!set initial condition
T = (b-a)*x + a

!set source function
S = S0*sin(pi*x)
```

Parallel Jacobi

Let's now parallelize the solver with OpenMP

- **Look for loops that can be parallelized**
- **Look for vectorized operations that can be converted to loops that can be parallelized**

Parallel:

```
!$omp parallel do
do i1=0,n+1
    x(i1) = i1*dx
    T(i1) = (b-a)*x(i1) + a !set initial condition
    S(i1) = S0*sin(pi*x(i1)) !set source function
end do
!$omp end parallel do
```

Parallel Jacobi notes

- **Will only see speedup with n > ~20000 (commonly seen in 2D problems)**
- **See *jacobi1s_omp.f90, jacobi1_omp.py***
- **f2py and OpenMP:** f2py --f90flags='-fopenmp' -lgomp -c jacobi1s_omp.f90 -m j1
- **On my laptop:**
  f2py --f90flags='-fopenmp' –L/usr/local/lib -lgomp -c jacobi1s_omp.f90 -m j1

Time-dependent problem

Today: 1-D problem

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + S(x,t)$$

$$T(x, t=0) = f(x)$$

$$T(x=0,t) = a(t), \;\; T(x=1,t) = b(t)$$

$$0 \le x \le 1$$

Simple (inefficient) approach: *method of lines*

1. **Discretize spatial variable → N+2 points beteween 0 and 1**

2. **Solve resulting N ODEs with solver of choice (*odeint*, *ode15s ,…)***

Time-dependent problem

Again, we discretize the derivative as:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$

$$x_i = i * \Delta x,\ i = 0, 1, 2, ..., N + 1$$

$$(N + 1) * \Delta x = 1$$

Time-dependent problem

Again, we discretize the derivative as:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$

$$x_i = i * \Delta x, \; i = 0, 1, 2, ..., N+1$$

$$(N+1) * \Delta x = 1$$

So, we have N ODEs:

$$\frac{dT_i}{dt} = S_i(t) + \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}, \; i = 1, 2, ..., N$$

with the boundary conditions substituted in the RHS when needed.

Solving single ODE in python

- **Use** *odeint* **from** *scipy.integrate* **module to solve:**

$$\frac{dy}{dt} = -ay$$

- **Basic idea: discretize time,** t = 0, dt, …, N*dt**, and starting from** y(0) **march forward in time and compute** y(dt), … y(N*dt)

- *odeint* **chooses the stepsize,** dt, **so that error tolerances are satisfied**

- **Need to specify:**
  - **Initial condition**
  - **Timespan for integration**
  - **A Python function which provides RHS of the ODE to odeint**

- **Look at** *ode_example.py* **and lab 4**

Time-dependent problem

Solving N ODEs:

$$\frac{dT_i}{dt} = S_i(t) + \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}, \; i = 1, 2, ..., N$$

- **Will need to provide *N* initial conditions when calling *odeint*.**
- **The python function which provides RHS to *odeint* will:**
  - **Take *t* and $T_1,...,T_N$ and any other needed parameters as input**
  - **Return *N* values for *dT/dt* as output**
- **No need for Fortran for 1D problems, but may be faster for two and three dimensions.**