Scientific computing needs supercomputers, but also something else!

Xing Cai

NUDT, March 29, 2012

Where am I from?

- Simula Research Lab
- University of Oslo

More about myself

Research interests:

- Methodologies for parallel programming
- High-performance scientific computing and applications
- Numerical methods for partial differential equations

Objectives for this visit:

- Strengthen collaboration with the MASA group at NUDT
  - learn about cutting-edge computer architectures
- Use high-end supercomputers for doing science

Today's talk

- Performance is a very vague concept
  - Hardware capability of a supercomputer

    $\nparallel$

  - Achievable performance of a software code implementing a particular numerical algorithm
- Important: Speed of data movement vs. speed of floating-point operations
  - We should be able to pinpoint the performance bottleneck
- How to choose a best-performing numerical algorithm?
  - It depends on many things: application, hardware, problem size
  - We want the fastest computing time, while securing a certain level of accuracy

Top500

Rank	Site	Computer
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu
2	National Supercomputing Center in Tianjin China	NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT
3	DOE/SC/Oak Ridge National Laboratory United States	Cray XT5-HE Opteron 6-core 2.6 GHz Cray Inc.
4	National Supercomputing Centre in Shenzhen (NSCS) China	Dawning TC3600 Blade System, Xeon X5650 6C 2.66GHz, Infiniband QDR, NVIDIA 2050 Dawning
5	GSIC Center, Tokyo Institute of Technology Japan	HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows NEC/HP

- `http://www.top500.org`
- Tianhe-1A: Pride of Chinese supercomputing and NUDT!
  - No.1 on Top500 list of November 2010
  - No.2 on Top500 list of June 2011
  - No.2 on Top500 list of November 2011
- Peak floating-point rate: 4.701 Peta Flops/second
- Linpack floating-point rate: 2.566 Peta Flops/second

About Linpack

- `http://www.top500.org/project/linpack`
- Solving a dense system of linear equations
- Double-precision operation count: $\frac{2}{3}n^3 + \mathcal{O}(n^2)$
- *Compute-bound*, not data-movement-bandwidth bound

Remarks so far

- Linpack shows idealized performance of a supercomputer
- But lots of applications depend more on how fast data is transferred:
  - *main memory* $\leftrightarrow$ *L3 cache* $\leftrightarrow$ *L2 cache* $\leftrightarrow$ *L1 cache* $\leftrightarrow$ *registers*
- There has been perhaps too much focus on achieving GFLOP/s, TFLOP/s, PFLOP/s, EFLOP/s . . .
- What really counts is to compute sufficiently accurate solutions using the shortest amount of time
  - Hardware matters
  - Software matters
  - Numerical algorithm also matters!

Balance is important

- Machine balance ratio:

$$\frac{\text{data traffic bandwidth [GWords/sec]}}{\text{peak performance [GFlops/sec]}}$$

- Code balance ratio:

$$\frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

- If the code-balance ratio is higher than machine-balance ratio, the code will be *data-traffic-bandwidth-bound*
- Otherwise, the code is *compute-bound*

Performance bottlenecks

- For non-compute-bound codes, people tend to put the blame on the main memory bandwidth
- However, the performance bottleneck can be somewhere else along the data movement path:
  - *main memory* $\leftrightarrow$ *L3 cache* $\leftrightarrow$ *L2 cache* $\leftrightarrow$ *L1 cache* $\leftrightarrow$ *registers*
  - If data reuse is good in the caches, data traffic volume is decreasing from *registers* to *main memory*
- For example, the bandwidth between the L1 cache and registers can be a bottleneck

Can we roughly predict the performance?

Yes, if we know for software,

- n_{FP} — # floating-point operations
- n_{LD} — # loads from L1 cache to registers
- n_{ST} — # stores from registers to L1 cache
- n^{M}_{2way} — # reads+writes between memory and last-level cache

and if we also know for hardware,

- F — peak floating-point capability
- B^{r}_{L1} — load bandwidth from L1 cache to registers
- B^{w}_{L1} — store bandwidth from registers to L1 cache
- B_M — 2-way bandwidth of main memory

Simplified prediction models

The case of using a single core:

$$\text{Time usage } = \max\left(\frac{n_{\mathrm{FP}}}{F}, \frac{n_{\mathrm{LD}}}{B^r_{L1}}, \frac{n_{\mathrm{ST}}}{B^w_{L1}}, \frac{n^M_{\mathrm{2way}}}{B_M}\right).$$

The case of using p cores:

$$\text{Time usage } = \max\left(\frac{n_{\mathrm{FP}}}{pF}, \frac{n_{\mathrm{LD}}}{pB^r_{L1}}, \frac{n_{\mathrm{ST}}}{pB^w_{L1}}, \frac{n^M_{\mathrm{2way}}}{B^p_M}\right).$$

Note: B^p_M does not scale linearly as $p \cdot B_M$

Simple example 1

The simplest 3D heat equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + f$$

Fully-explicit numerical scheme ($\Delta x = \Delta y = \Delta z = h$):

$$\frac{u_{i,j,k}^{\ell+1} - u_{i,j,k}^{\ell}}{\Delta t}$$

$$= \frac{u_{i-1,j,k}^{\ell} + u_{i,j-1,k}^{\ell} + u_{i,j,k-1}^{\ell} - 6u_{i,j,k}^{\ell} + u_{i+1,j,k}^{\ell} + u_{i,j+1,k}^{\ell} + u_{i,j,k+1}^{\ell}}{h^2} + f_{i,j,k}$$

Simple example 1 (cont'd)

The C code:

```
   t = 0.;
   while (t<T) {
#pragma omp for private(i,j) schedule(static)
     for (k=1; k<n-1; k++)
       for (j=1; j<n-1; j++)
         for (i=1; i<n-1; i++)
           u_new[k][j][i] = u_old[k][j][i] + rhs[k][j][i]
             + factor*(u_old[k][j][i-1]+u_old[k][j][i+1]
                       +u_old[k][j-1][i]+u_old[k][j+1][i]
                       +u_old[k-1][j][i]+u_old[k+1][j][i]
                       -6*u_old[k][j][i]);

#pragma omp single
     {
       /* pointer swap */
       /* ... */
       t += dt;
     }
   }
```

Simple example 1 (cont'd)

Performance study on a computer with two quad-core Intel Xeon 2.0 GHz E5504 CPUs

- $F = 4$ GFLOP/s (for a non-SIMD compiler)
- $B^r_{L1} = B^w_{L1} = 16$ GB/s
- B^p_M values are measured by the STREAM benchmark
- Per time step and per grid point, $n_{\mathrm{FP}} = 10$, $n_{\mathrm{LD}} = 11 \times 8$ bytes, $n_{\mathrm{ST}} = 1 \times 8$ bytes, $n^M_{\mathrm{2way}} = 3 \times 8$ bytes

# cores	1	2	4	6	8
B^p_M	6.22 GB/s	12.19 GB/s	13.89 GB/s	13.24 GB/s	13.03 GB/s
T_A	358.32 s	184.84 s	120.72 s	114.61 s	122.14 s
T_P	320.20 s	160.10 s	100.59 s	105.53 s	107.23 s

T_A: actual time usage, T_P: predicted time usage

points: $99 \times 99 \times 99$, # time steps: 60001

Simple example 2

3D heat equation with variable coefficient:

$$\frac{\partial u}{\partial t} = \nabla \cdot (\kappa \nabla u) + f$$

Fully-explicit numerical scheme ($\Delta x = \Delta y = \Delta z = h$):

$$\begin{aligned}
&\frac{u^{\ell+1}_{i,j,k} - u^{\ell}_{i,j,k}}{\Delta t} \\
= &\frac{1}{2h^2}\big((\kappa_{i+1,j,k} + \kappa_{i,j,k})(u^\ell_{i+1,j,k} - u^\ell_{i,j,k}) - (\kappa_{i,j,k} + \kappa_{i-1,j,k})(u^\ell_{i,j,k} - u^\ell_{i-1,j,k}) \\
&+ (\kappa_{i,j+1,k} + \kappa_{i,j,k})(u^\ell_{i,j+1,k} - u^\ell_{i,j,k}) - (\kappa_{i,j,k} + \kappa_{i,j-1,k})(u^\ell_{i,j,k} - u^\ell_{i,j-1,k}) \\
&+ (\kappa_{i,j,k+1} + \kappa_{i,j,k})(u^\ell_{i,j,k+1} - u^\ell_{i,j,k}) - (\kappa_{i,j,k} + \kappa_{i,j,k-1})(u^\ell_{i,j,k} - u^\ell_{i,j,k-1})\big) \\
&+ f_{i,j,k}
\end{aligned}$$

Simple example 2 (cont'd)

- Per time step and per grid point, $n_{\mathrm{FP}} = 26$, $n_{\mathrm{LD}} = 21 \times 8$ bytes, $n_{\mathrm{ST}} = 1 \times 8$ bytes, $n^{M}_{\mathrm{2way}} = 4 \times 8$ bytes

# cores	1	2	4	6	8
T_A	648.07	332.28	180.82	156.15	169.16
T_P	611.30	305.65	152.82	140.71	142.98

T_A: actual time usage, T_P: predicted time usage

points: $99 \times 99 \times 99$, # time steps: 60001

Simple example 3

Sparse matrix-vector multiply:

$$y = Ax$$

where each row of matrix A has 7 nonzeros, due to finite differencing

- Compressed sparse row storage of matrix A
- Per matrix row $n_{\mathrm{FP}} = 14$, $n_{\mathrm{LD}} = 14 \times 8 + 9 \times 4$ bytes, $n_{\mathrm{ST}} = 1 \times 8$ bytes, $n^{M}_{\mathrm{2way}} = 9 \times 8 + 8 \times 4$ bytes

# cores	1	2	4	6	8
T_A	0.020586	0.012319	0.012163	0.012976	0.013870
T_P	0.016720	0.008532	0.007487	0.007855	0.007982

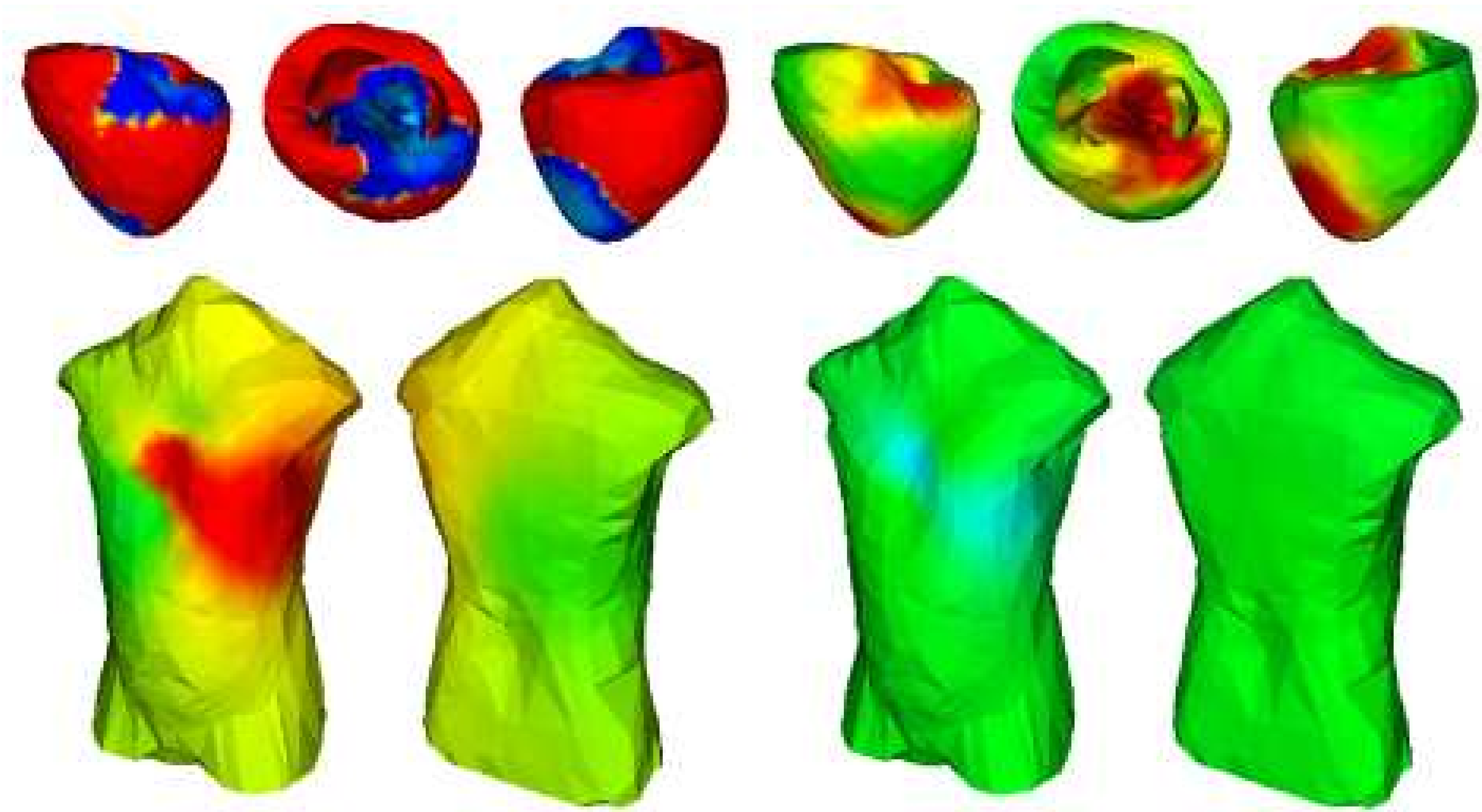
T_A: actual time usage, T_P: predicted time usage

rows in A: 10^6

Remarks

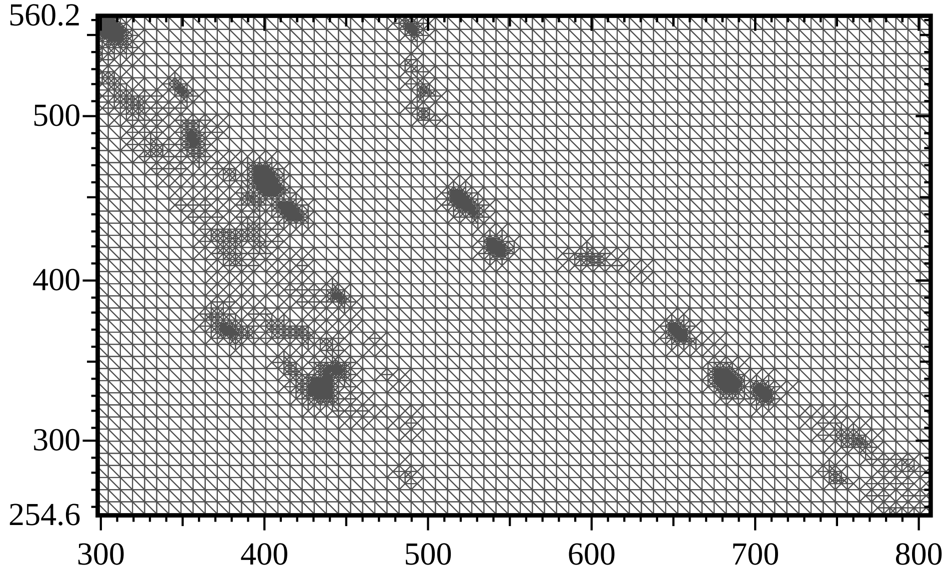
- For structured-grid applications, data reuse in cache is considerable, therefore relieving the pressure on the main memory
- For typical scientific codes, the machine balance (even considering the L1 cache bandwidth) is lower than the code ratio
- When a small number of cores are in use per CPU
  - L1 cache bandwidth may be the performance bottleneck
- When all the cores are in use
  - Main memory is likely the bottleneck, because main memory bandwidth doesn't scale
- For unstructured-grid applications, the main memory bandwidth is the most likely bottleneck
- Machine balance ratio will possibly continue to decrease in future

Challenge 1: unstructured mesh



- Unstructured computational meshes are important in many situations
- Increased ratio of code balance (due to complex data layout)

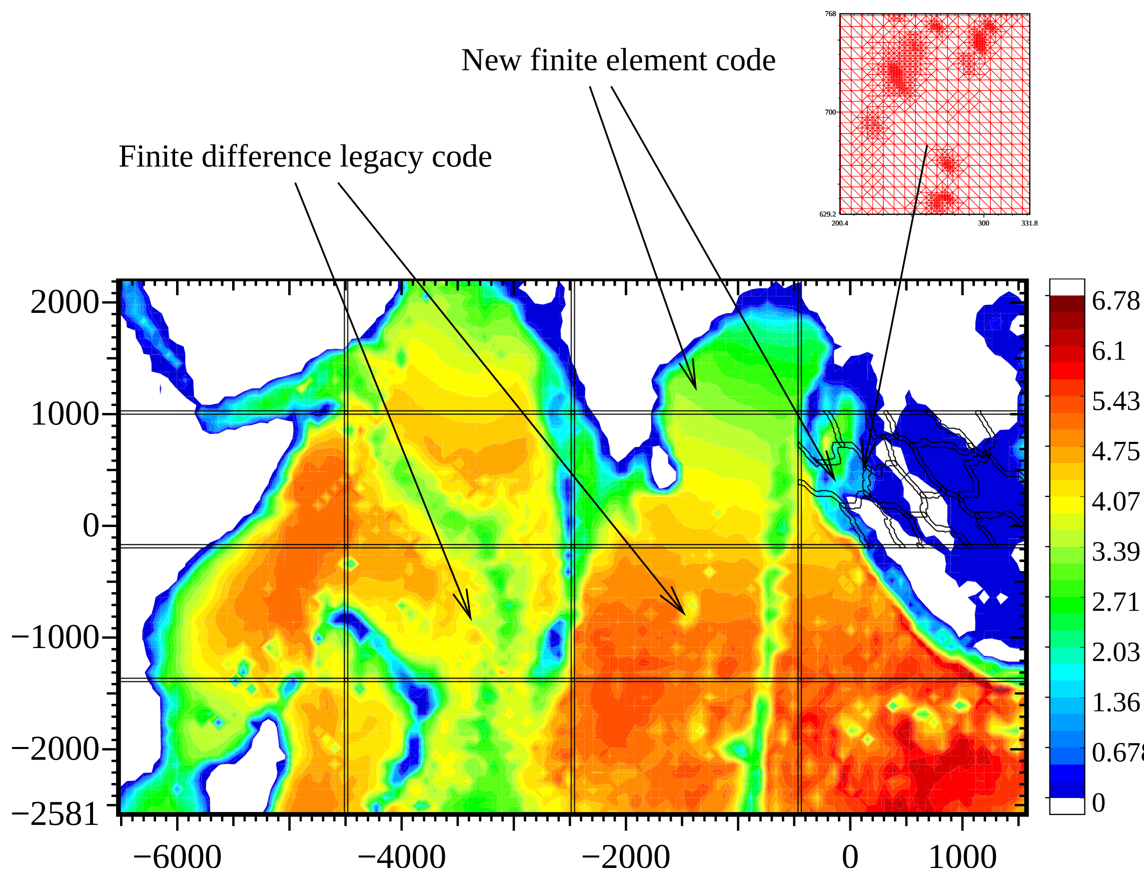
Challenge 2: adaptive mesh refinement



- Can reduce the overall computational complexity
- More complex data layout $\Rightarrow$ even higher code balance ratio

More challenges for floating-point rates

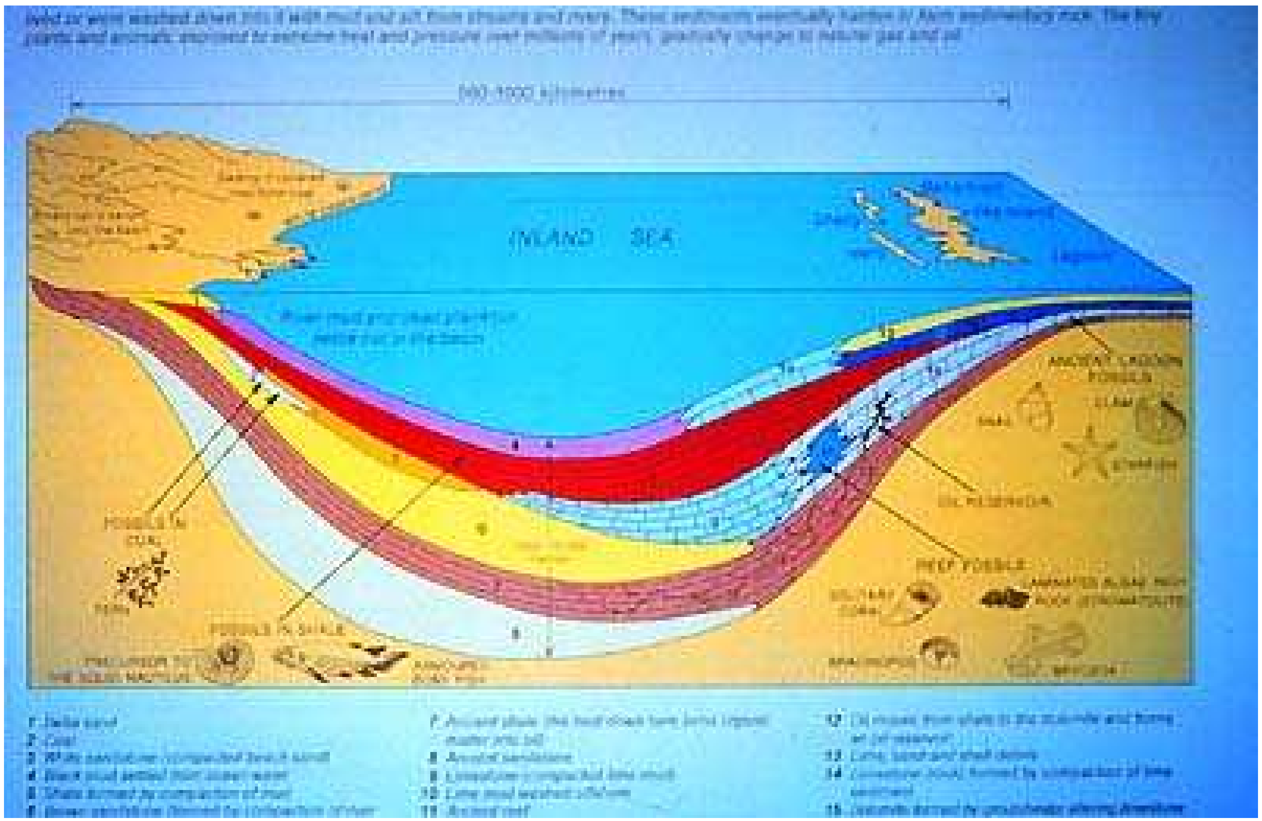
- Sophisticated preconditioner (e.g. algebraic/geometric multigrid) for solving linear systems
- Parallel hybrid software code encompassing very different subdomains



A real-world case

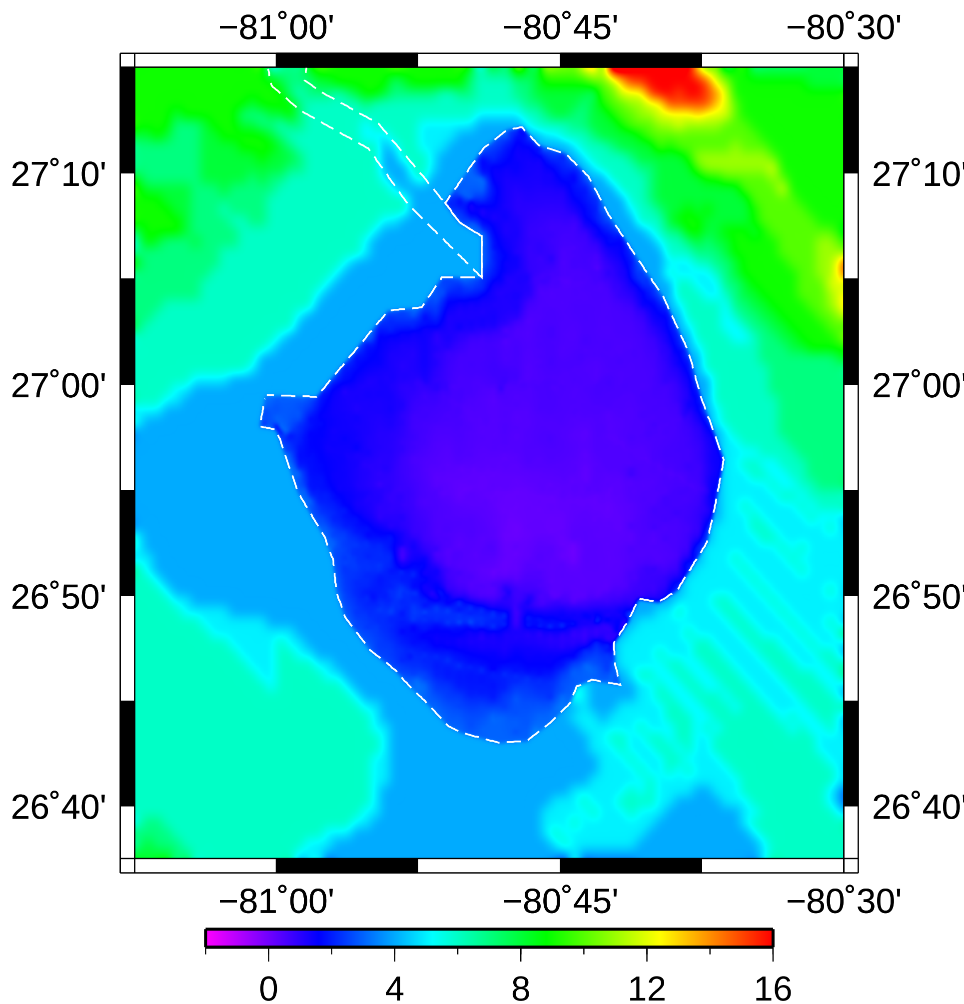
How to efficiently simulate sedimentary basin formation?

- Numerical methods are different in accuracy, stability and speed
- The choice of a best method is problem dependent



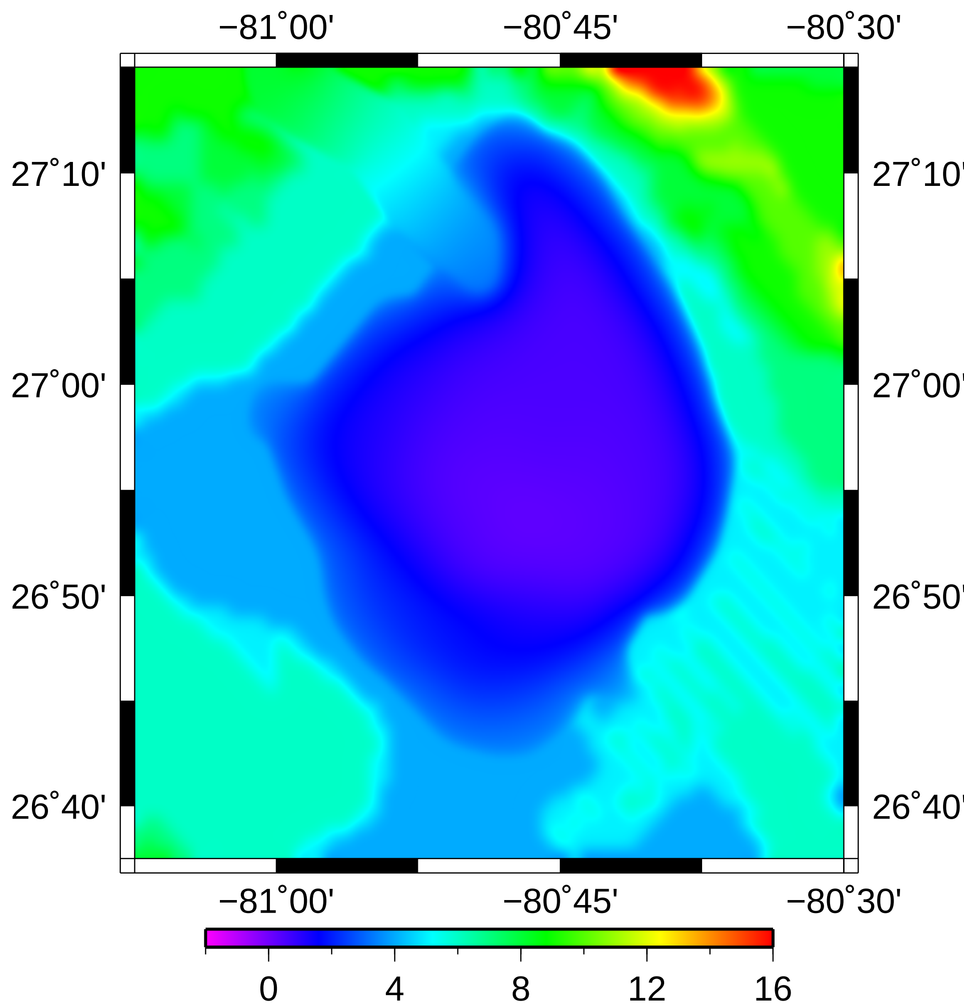
From `http://www.lithoprobe.ca/`

Simulation snapshot 1



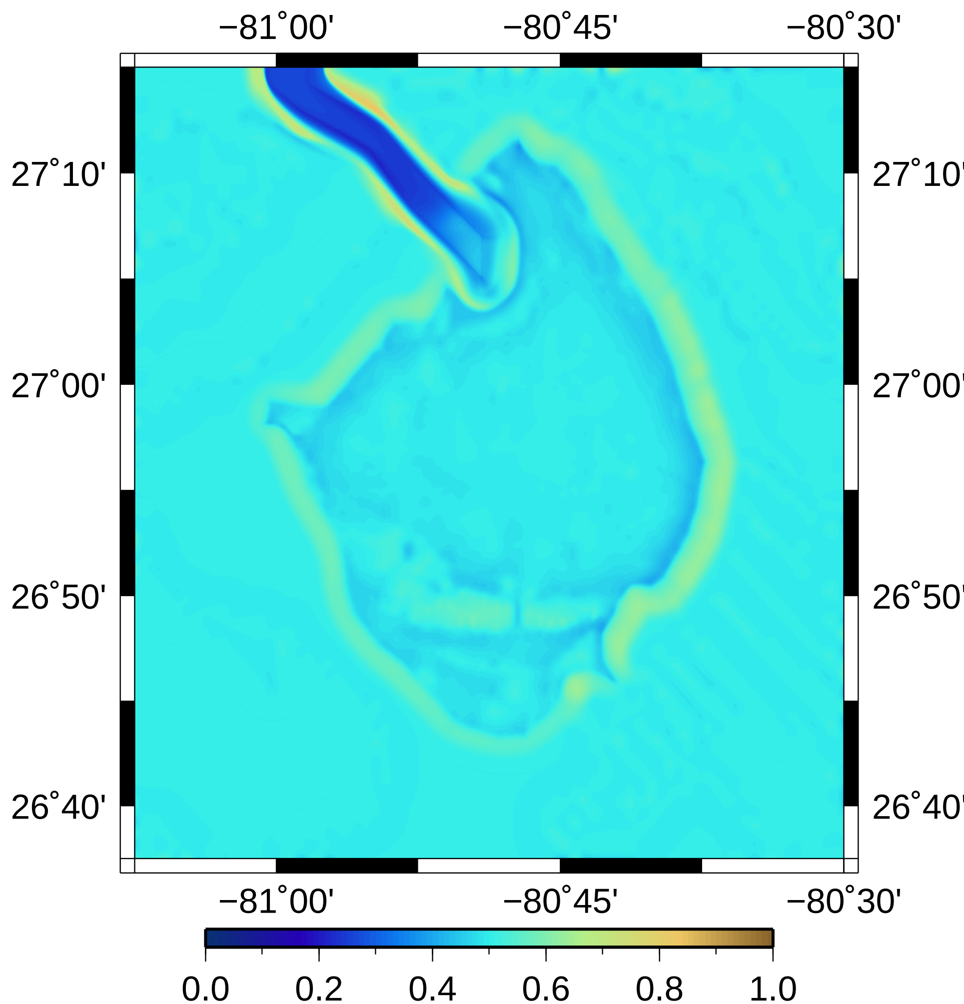
Lake Okeechobee, Florida; Initial $h(x, y)$

Simulation snapshot 2



Lake Okeechobee, Florida; Solution of $h(x, y)$ after 1,000 years

Simulation snapshot 3



Lake Okeechobee, Florida; Solution of $s(x, y)$ after 1,000 years

The math model

Two coupled nonlinear partial differential equations:

$$\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla \cdot (\alpha s \nabla h) + \frac{1}{C_m}\nabla \cdot (\beta(1-s)\nabla h), \tag{1}$$

$$A\frac{\partial s}{\partial t} + s\frac{\partial h}{\partial t} = \frac{1}{C_s}\nabla \cdot (\alpha s \nabla h). \tag{2}$$

- Two lithologies (sand and mud) are considered in sedimentary basin filling
- $h(x, y, t)$ — height of basin surface
- $s(x, y, t)$ — fraction of sand

Choices of numerical methods

There are at least five temporal discretization schemes:

- Fully-explicit (first-order accuracy)
  - Update h and s separately, no need to solve linear systems
- Semi-implicit 1 (first-order accuracy)
  - Update h and s separately, solving one linear system for h, and one for s per time step
- Semi-implicit 2 (second-order accuracy)
  - Update h and s separately, solving a linear system for h twice, and twice s per time step
- Fully-implicit 1 (first-order accuracy)
  - *Backward Euler*: Update h and s simultaneously, need Newton iterations per time steps
- Fully-implicit 2 (second-order accuracy)
  - *Crank-Nicolson*: Update h and s simultaneously, need Newton iterations per time steps

Comparison of the five schemes

Scheme	Action	n_{FP}	n_{LD}	n_{ST}	n^M_{2way}
Fully -explicit	Compute h	57	43	1	5
	Compute s	37	24	1	5
Semi-implicit 1	Set up h system	62	21	8	10
	Solve h system	15	52	9	21
	Set up s system	35	35	10	10
	Solve s system	15	68	14	21
Semi-implicit 2	Set up h system	262	158	24	20
	Solve h system	15	52	9	21
	Set up s system	117	125	29	20
	Solve s system	15	68	14	21
Fully-implicit 1	Set up h-s system	150	150	92	27
	Solve h-s system	46	264	52	62
Fully-implicit 2	Set up h-s system	225	223	138	28
	Solve h-s system	46	264	52	62

Preliminary measurements on Tianhe

- Spatial mesh resolution: 9206×6108
  - Fully-explicit using $\Delta t = 0.005$ (due to strict stability limit)
  - Semi-implicit 1 using $\Delta t = 10$
  - Semi-implicit 2 using $\Delta t = 0.25$ (due to stability requirement)

Scheme	Measurement	240 cores	480 cores	960 cores	1920 cores
Fully-	Time	150.76	78.13	36.57	17.92
explicit	GFLOP/s	701.20	1353.04	2890.70	5899.16
Semi-	Time	131.77	70.70	30.45	18.02
implicit 1	GFLOP/s	30.90	57.59	133.72	225.97
Semi-	Time	648.98	356.87	137.54	78.82
implicit 2	GFLOP/s	51.03	92.81	240.80	420.19

Into the era of GPU computing

Example of CPU peak performance versus GPU peak performance:

- Xeon 6-core X5670
  - Peak Double-precision floating-points: $6 \times 4 \times 2.93 = 70.32$ GFLOP/s
  - Peak Memory bandwidth 64-bit (to the L3-cache): $3 \times 8 \times 1.333 = 32$ GB/s
- Tesla M2050
  - 512 CUDA cores
  - Peak Double-precision floating-points: 515 GFLOP/s
  - Peak memory 384-bit bandwidth: 148 GB/s

Implications of GPU computing

- Ratio of $\frac{\text{memory bandwidth}}{\text{floating-point rate}}$ decreases further
- Compute-bound algorithms are favored on GPUs
- Simple algorithms are favored on GPUs
  - Advanced algorithms may not suit on GPUs

# GPUs	GFLOP/s
16	542.35
32	828.94
64	1591.53
128	1855.36
256	3365.59

Preliminary runs of the fully-explicit scheme on the GPUs of Tianhe

Summary

1. Utilizing the full potential of a supercomputer is challenging
   - because computations may very well be bound by data-traffic bandwidth
2. It is possible to (roughly) predict the minimum required computing time for well-defined computations
   - even before the software code is written
3. When the goal is to solve a scientific problem fastest possible, with sufficient accuracy:
   - choosing a best-performing numerical algorithm depends on many factors (e.g. ratio of code balance, problem size, hardware)
4. On GPUs, the simplest numerical algorithms with high computational intensity *may* beat advanced algorithms with low computational complexity

Acknowledgments

Thanks to my collaborators:

- Wenjie Wei (Simula Research Lab)
- Dr. Stuart Clark (Simula Research Lab)
- Prof. Zhang Chunyuan (MASA/NUDT)
- Prof. Wen Mei (MASA/NUDT)
- Prof. Wu Nan (MASA/NUDT)
- Su Huayou (MASA/NUDT)
- Chai Jun (MASA/NUDT)