4: Some Model Programming Tasks

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Parallel Programming

.....

https://people.sc.fsu.edu/~jburkardt/presentations/...
parallel_tasks_2008_vt.pdf

10-12 June 2008

Model Programming Tasks

We need to consider some simple computational problems which

- occur frequently in scientific programming;
- are simple enough to analyze and understand;
- illustrate issues in parallel programming.

Model Programming Tasks

- *Toys:* **SUM**, **SORT**
- *Monte Carlo:*, **INTEGRATION**, **ISING**, **RADIATION**
- *Differential:* **DE**, **HEAT**, **NBODY**
- *Searches:* **LIST**, **SPACETRIAL**, **RANDOM**
- *Linear Algebra:* **DOT, SAXPY, MV, MM**, **JACOBI**, **SGEFA**

Toy Problems

Here are some tasks we can imagine carrying out using just a few variables, and a few lines of code:

- **SUM**, sum a list of 10,000 numbers
- **SORT**, sort a list of 10,000 numbers

If we think about doing these tasks in parallel, we can begin to see some of the issues.

Toy Problems: The SUM Task

The SUM task: compute the sum of 10,000 numbers.

We'll assume the numbers are available in a vector called **x**:

sum = x[0] + x[1] + x[2] + ... + x[9999]

It's easy to see that having more processors **might** help. Every + sign is a task. But where do the individual temporary results go? How do we combine them?

Once multiple processors are involved, computation must include **communication**!

Toy Problems: The SUM Task in Shared Memory

The teacher writes down **S**=0 on a piece of paper, the current sum.

She has a box of numbers **x[0]**, **x[1]**,.. **x[9999]**.

Each student may come to the front, copy the current value of **S** and take one number **x[i]** from the box. The student goes back to a desk, computes **S+x[i]**, and comes back to the teacher, erases the current value of **S** and writes the new one

What could go wrong here?

"It worked fine when I tested it using just 1 student!"

Toy Problems: The SUM Task in Distributed Memory

The teacher writes down **S**=0 on a piece of paper, the current sum.

She has a box of numbers **x[0]**, **x[1]**,.. **x[9999]**.

She divides her list of numbers up into roughly equal parts, calls each student by name and reads out a list of numbers for that student to add up.

Each student sums up their numbers, and then whenever they finish, they shout out their result.

The teacher tries to hear each result and add it to the running total.

What could go wrong here?

"I think I didn't hear some of the results!"

Toy Problems: The SUM Task (Observation)

In both versions of our parallel SUM task, it was clear that having student help was a plus, but that there were new issues of distributing the work and collecting the results.

Moreover, it should be clear that a clumsy or inefficient communication of data and results could slow down the calculation, or even make it incorrect.

Toy Problems: The SORT Task

The SORT task: sort 10,000 numbers.

It's easy for one person (or processor) to sort data.

There are fast ways (quicksort, heapsort) and slow ways (bubblesort), but they all get there in the end.

When we have several people available to sort one list, can we actually take advantage of the extra help?

Toy Problems: The SORT Task (merge)

If I had 10,000 cards to sort, I'd certainly be glad for help!

If I split the cards among the helpers, and each helper sorts their deck, then I have a fairly easy job at the end of **merging** the sorted subdecks into one sorted deck.

Of course, I won't be doing the merging part in parallel.

Toy Problems: The SORT Task (buckets)

A "bucket sort" keeps all the processors busy. We assume we're sorting real numbers between 0 and 1. We imagine the data range as being divided up into **N** equal subranges or buckets.

Each processor is given a share of the data, and tosses each data value it has into the appropriate bucket. When all the data is in buckets, processor **I** sends the first bucket to the first processor, the second bucket of data to the second processor, and so on.

In the final stage, each processor sorts its share of the data,

The data is completely sorted, although distributed across the processors.

Communication issues? Efficiency? Worst case?

Toy Problems: The SORT Task (observations)

What works sequentially for sorting doesn't work in parallel. To get the benefits of parallel processing, we may need to modify our algorithm, or even choose a new one!

A feature of the **SORT** task is conditional logic We might say:

if X(I) $<$ X(J) **then** interchange X(I) and X(J)

In basis sorting algorithms, you want to compare one data item with any other data item in the whole array, then maybe swap them.

This has to be done carefully in shared or distributed memory.

Also, because our operations are now **conditional**, it can be harder for the compiler to help in optimization or parallelization.

Monte Carlo Problems

Monte Carlo problems typically involve many instances of the same calculations, done with randomly selected input. In some cases, the exact same procedure is done each time; in others, different inputs can result in very different calculations.

- **RANDOM**, generate 10,000 random numbers
- **INT**, Monte Carlo integration
- **ISING**, Monte Carlo magnetism simulation
- **RADIATION**, Monte Carlo radiation simulation

Monte Carlo: The RANDOM Task

The basic process needed by all Monte Carlo simulations is to generate a lot of **"random" numbers**;

Two common generators in use:

- **LC-RNG**, Linear congruential random number generator
- **LF-RNG**, Lagged Fibonacci random number generator

The user influences the random number generator by supplying a "seed" or calling a seed routine or allowing a default seed.

Monte Carlo: The Random Task

The easiest way to get random numbers is to call the system random number generator.

But on distributed memory systems, you need to make sure you use a different seed on each processor, otherwise you may end up computing N copies of the same random values.

On shared memory systems, there will be several threads running in the same memory space. You will have to experiment to see whether your system's random number generator works in a way that allows it to be "shared" or not. You may need to set up a different seed for each thread.

Monte Carlo: The RANDOM Task

The LC-RNG really operates on a string of integers, and uses a simply formula to produce a corresponding real number between 0 and 1. The user gives the first integer, **I0**, which is called *the seed*. The calculation is then:

```
I0 = ( given by user )
I1 = ( A * I0 + B ) mod C   ==>  X1 = ( I1 / IMAX )
I2 = ( A * I1 + B ) mod C   ==>  X2 = ( I2 / IMAX )
I3 = ( A * I2 + B ) mod C   ==>  X3 = ( I3 / IMAX )
```

The user can reset the seed at any time, which will cause the random number sequence to jump back or ahead.

Monte Carlo: The RANDOM Task

On a shared memory system with N threads, how do you compute N "next" random values?

It's possible, once you know the value of N, to compute new values AN, BN and CN which jump the old sequence ahead by N steps.

To get N seeds for this new sequence, start with I0, and compute I1, I2, up to I(N-1) using the original sequence.

Now give each processor its own seed, and use the new sequence.

Result: N sequences that don't overlap, and are simply the same sequence gotten when using a single processor.

Monte Carlo: The RANDOM Task

We use the old generator, starting with I0, to get the starting values I1, I2, I3. Then we use the new generator, which "hops" by 4's, so that we can quickly generate new values.

```
processor 0:  I0 => I4 => I8  => I12 => ...
processor 1:  I1 => I5 => I9  => I13 => ...
processor 2:  I2 => I6 => I10 => I14 => ...
processor 3:  I3 => I7 => I11 => I15 => ...
```

Monte Carlo: The Integration Task

The MC-INT task: estimate $\int_a^b f(x) \quad dx$.

The integral can be estimated by evaluating $f(x)$ at points randomly selected in $[a, b]$, and multiply the average value by the length of the interval $b - a$.

A general multidimensional integral over D can be estimated in the same way, by evaluating $f(x, y, z)$ at points randomly selected from D, and multiplying the average value by the volume of D.

Monte Carlo: The Integration Task

Interesting features:

works the same in any spatial dimension.

works the same for irregular regions.

the computation includes an estimate of its accuracy.

no matter how many samples have been taken, the user can stop with a valid estimate, or take more samples for an improved estimate.

the computation is *embarassingly parallel*.

Monte Carlo: The Ising Model

To understand an observed physical process, it is helpful to find a simplified model that seems to have some of the same properties.

In the Ising model of magnetism, each cell in a rectangular region is assigned an initial spin $+1$ or -1.

The total magnetism of the system is the sum of the spins.

In the ferromagnetic model, the energy is the negative of the sum of neighboring spins:

$$E = -\sum_{i,j} (s_{i,j}s_{i,j+1} + s_{i,j}s_{i+1,j})$$

(This clever sum counts all neighbor pairs just once, assuming wraparound at the boundaries.)

Monte Carlo: The Ising Model

Start with random spins in each cell, and total energy E.

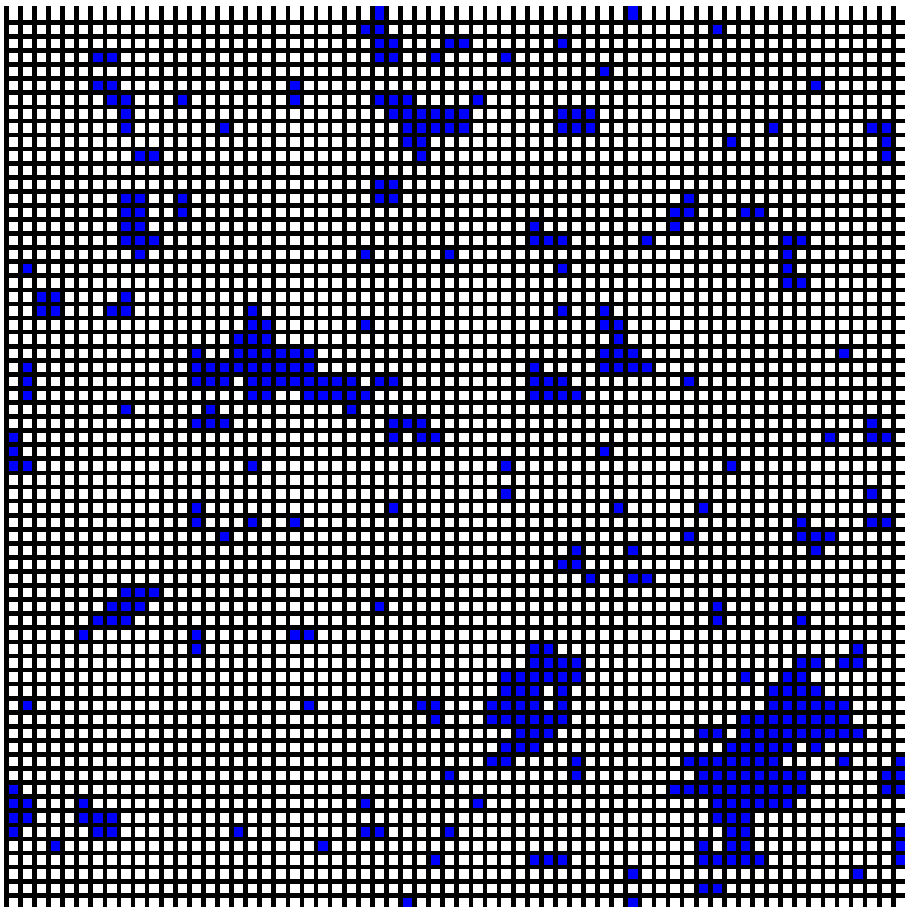
Let T be a nonnegative "temperature"

Choose a cell at random. Compute E_2, the energy the system would have if we reversed this single spin.

- If $E_2 < E$, reverse the spin always.
- If $E < E_2$, reverse the spin with probability $e^{\frac{-(E_2-E)}{T}}$.

For $0 < T$, the system will organize itself into regions of positive and negative spin.

Monte Carlo: The Ising Model



Monte Carlo: A Radiation Model

Neutrons emitted at origin with random energy E.

A shield wall of thickness T.

A neutron will penetrate D units of shield, where D depends on energy.

If still inside shield, a neutron is either absorbed (end of story) or re-emitted with random direction and lower energy.

For a given level of radiation (say 10,000 particles a second), and shield thickness T, what portion of the initial energy penetrates the shield?

The Monte Carlo method was created because of the necessity of making estimates of this kind!

Differential Problems

Differential equations describe changes over time in physical quantities. They can apply to the motion of a single point, the interaction of particles, or the "flow" of heat.

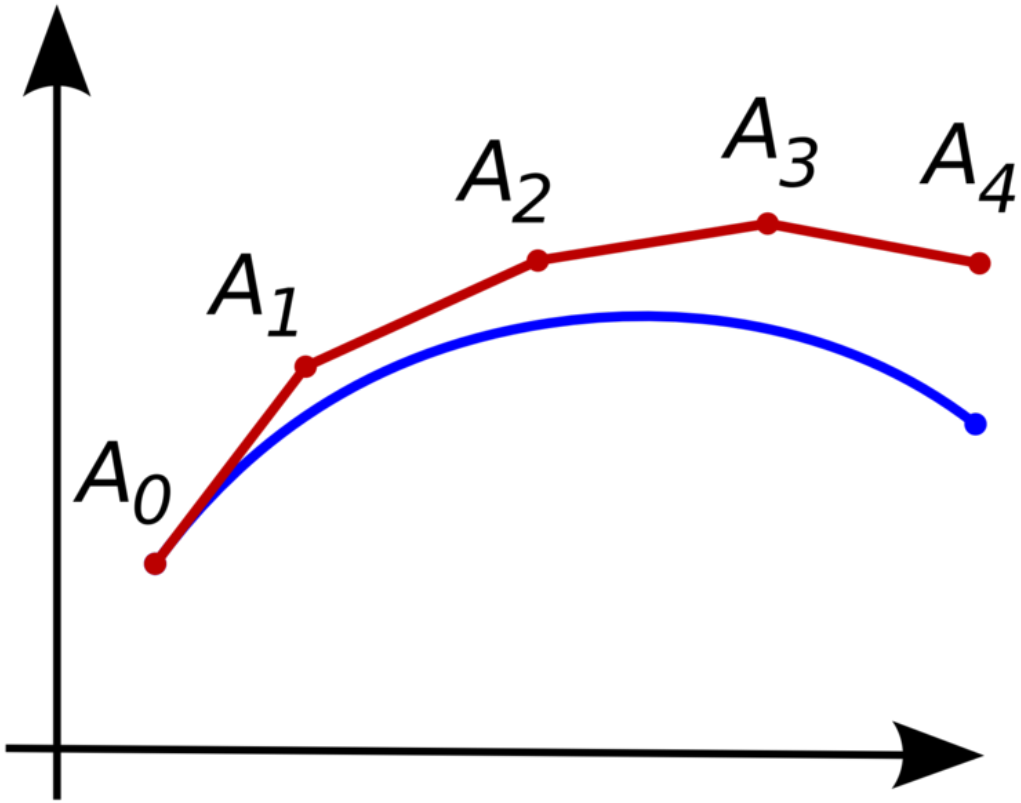
- 1 ODE, solve one ordinary differential equation
- HEAT, model heat flow over time on a wire
- NBODY, N-body evolutions (molecules or stars)

Differential Problems: 1 ODE

Determine the values of $x(t)$ over a range $t_0 <= t <= t_1$ given an initial value $x(t_0)$ and a differential equation

$$\frac{dx}{dt} = f(x, t)$$

Differential Problems: 1 ODE



A differential equation is (approximately) solved one step at a time! We can't start step 2 until we are completely done step 1.

Differential Problems: 1 ODE - (Observations)

Because a differential equation is solved one step at a time, it resists parallelization.

The solution data is like a list of values, where the $n+1$-th value depends on the n-th value.

Luckily, most differential equations actually have a substantial spatial component as well. They describe the behavior of many particles, or of an entire region. The spatial relationships can be handled in parallel.

Differential Problems: HEAT

The HEAT task: determine the values of $H(x,t)$ over a range $t_0 <= t <= t_1$ and space $x_0 <= x <= x_1$, given an initial value $H(x,t_0)$, boundary conditions, a heat source function $f(x,t)$, and a partial differential equation

$$\frac{\partial H}{\partial t} - k\frac{\partial^2 H}{\partial x^2} = f(x,t)$$

Parallel processing is actually useful for this problem!

Differential Problems: HEAT

Choose a set of nodes in the X direction.

Assign the initial value of H at all the nodes.

Rewrite the differential equation at node i:

- replace $\frac{\partial^2 H}{\partial x^2}$ by $\frac{(H(i-1)-2H(i)+H(i+1))}{\Delta x^2}$.
- replace $\frac{\partial H}{\partial t}$ by $\frac{HNEW(i)-H(i)}{\Delta t}$

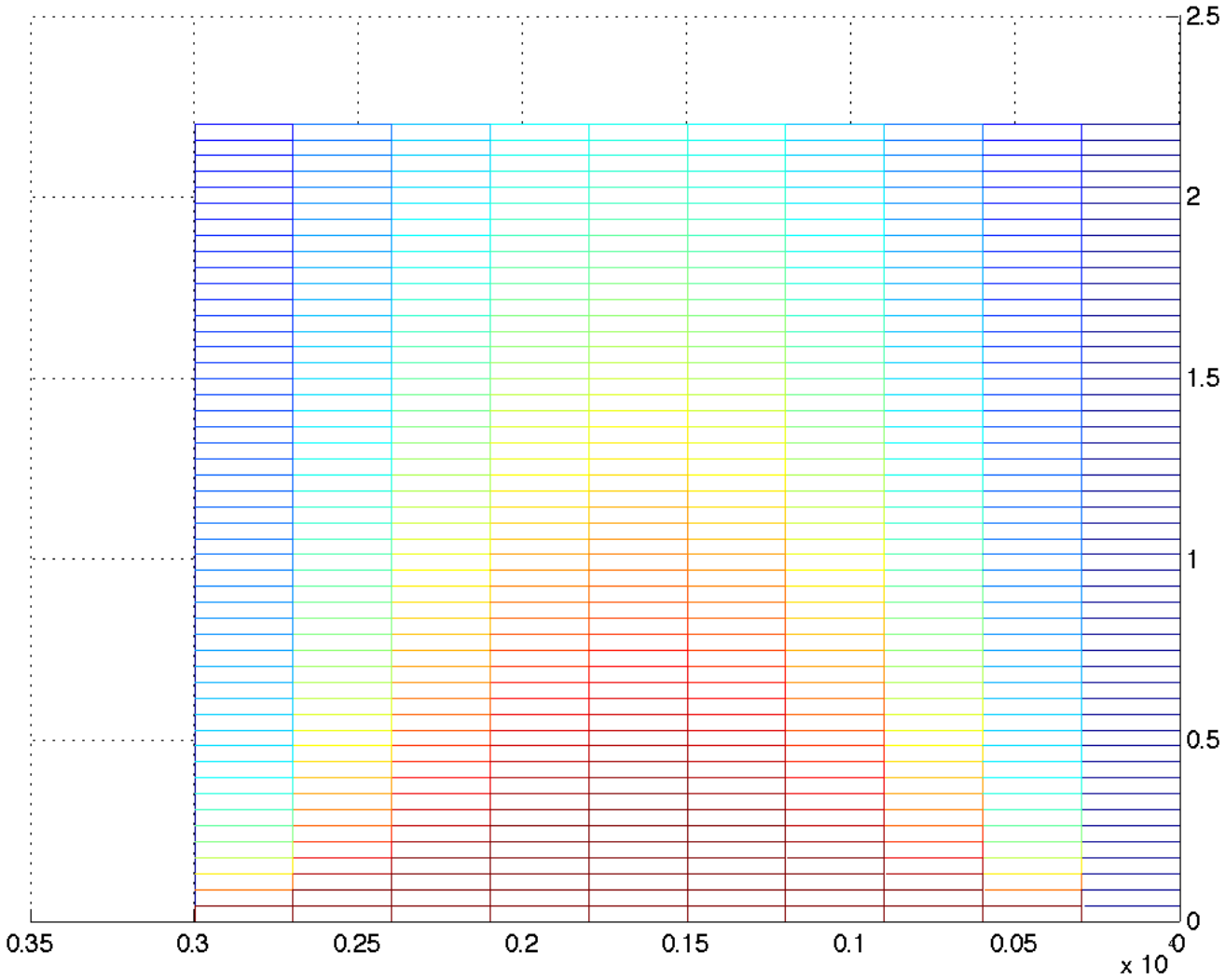
Differential Problems: HEAT

Now, at each node, you have a formula for computing the new value of $H(i)$ based on data at the old time.

The computations of the values at the next time are independent, and can be done in parallel.

For a distributed memory version, perhaps using domain decomposition, each processor lets its two neighbors know the updated internal boundary values.

Differential Problems: HEAT



Differential Problems: N-BODY

The **N Body** task:

- initial locations (molecules or stars)
- properties (mass, charge)
- velocities

Apply forces:

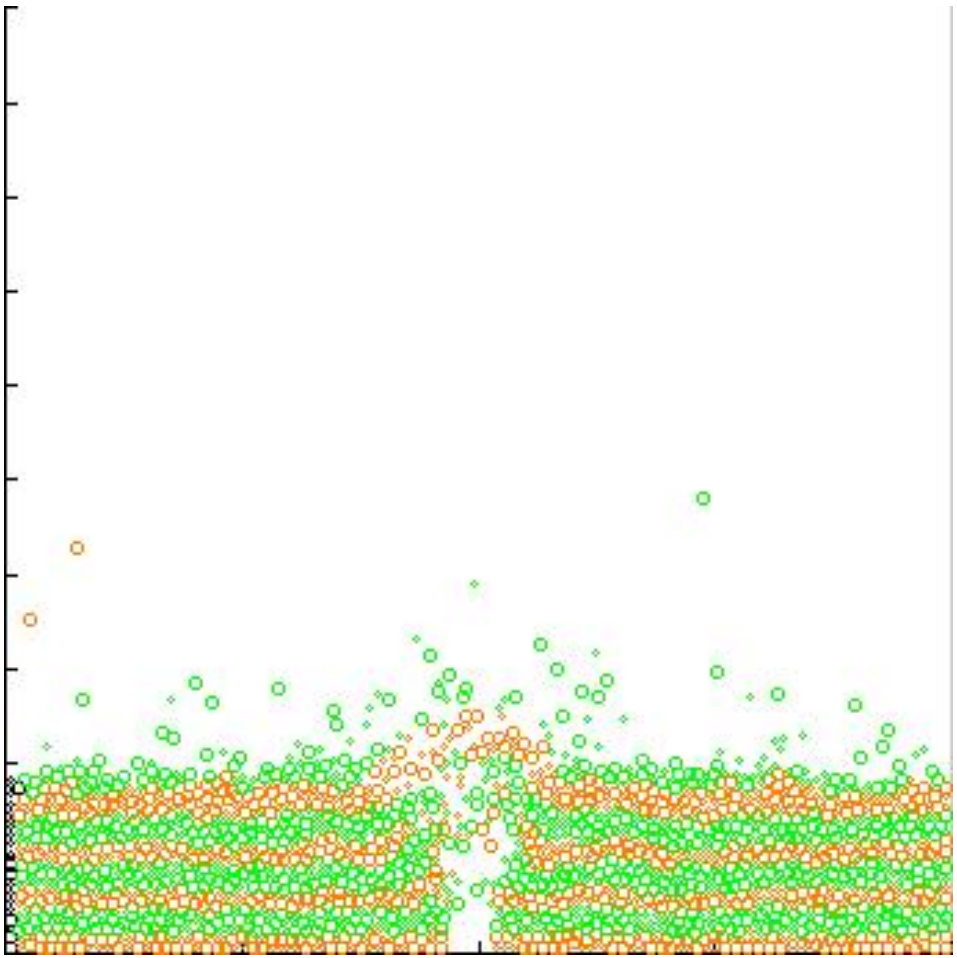
- gravity
- electric field
- van der Waals attraction

Update locations and velocities.

Parallelizable, but need to communicate new locations frequently.

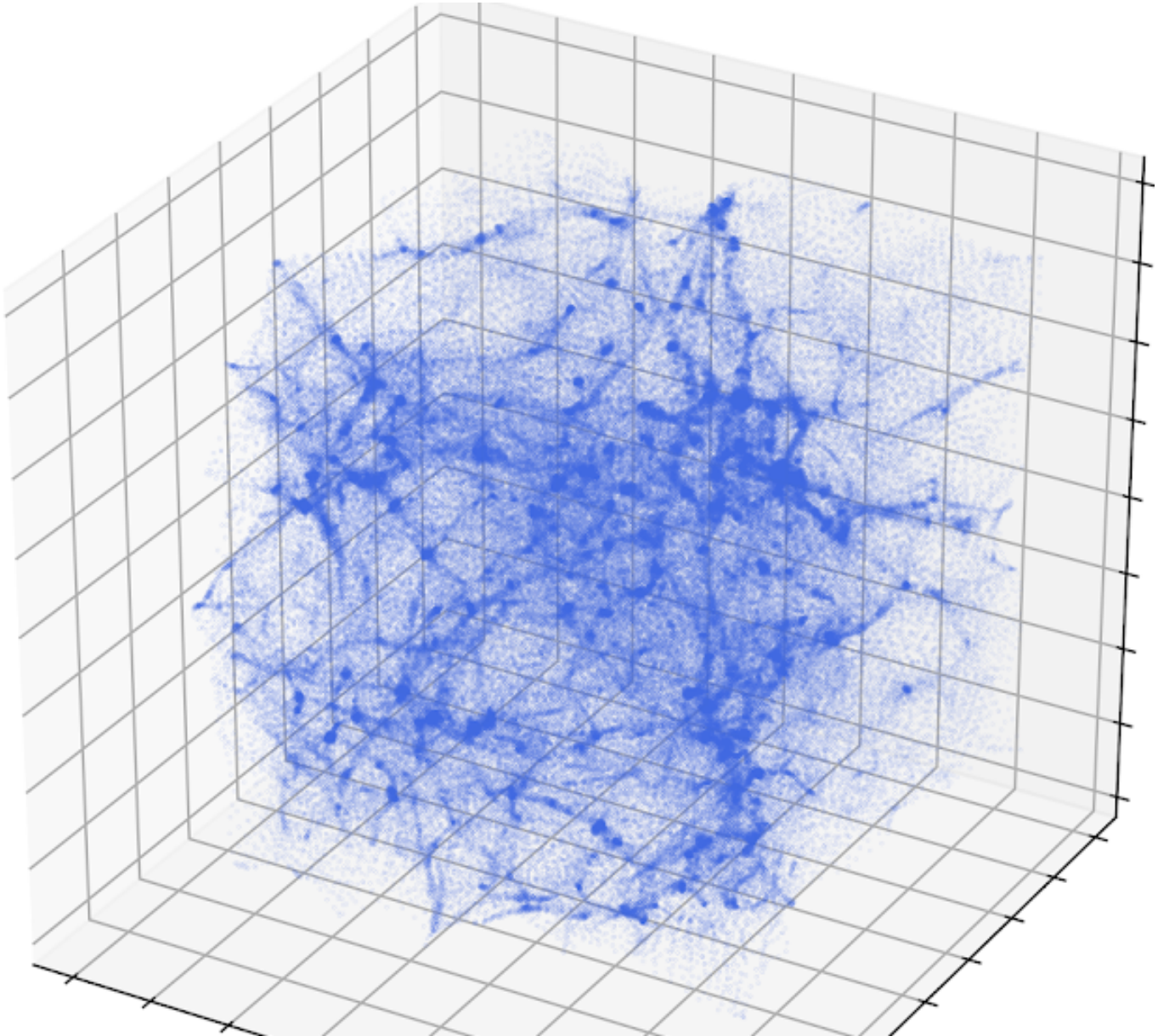
Differential Problems: N-BODY

This particle system simulates the process by which high pressures opened volcanic vents in layered strata.



Differential Problems: N-BODY

Here is an N-Body simulation which represent stars in colliding galaxies.



Search Problems

In these search problems, every search is "independent". There is almost no information we can use from one search to help us in the next one.

We can catalog searches by how hard it is to find the data.

- **SEARCH LIST**, select items from given list
- **SEARCH SPACE**, select items from a simple space
- **SEARCH TRIAL**, items must be generated by trial and error
- **SEARCH RANDOM**, items are found using random starting points

Search Problems: SEARCH LIST

Given a vector X of 1,000,000 values:

- find all occurrences of 7.5
- find the maximum value
- find the longest string of sorted values (careful!)

Search Problems: SEARCH SPACE

In these search problems, the data is not given beforehand, but there is a simple procedure for generating each data item.

- how many points with integer coordinates are contained in a 10-dimensional ellipse
- find all arrangements of the 32 chess pieces so neither side can move
- find all boolean inputs that make a logical function TRUE
- given a set of words, write them in any order. Shorten the list by merging repeated letters. Find the shortest list
- given a function F(X,Y), find a path from (X1,Y1) to (X2,Y2) that minimizes the line integral of F

Search Problems: SEARCH TRIAL

In these search problems, it's not clear how many data items there are. One way to generate them is by trial and error, or backtracking. If you manage to generate an item, then you can "search" it, that is, see if it's one you wanted.

- find all arrangements of 8 nonattacking chess queens
- given a knapsack that can hold no more than 20 pounds, and a collection of objects of varying weights and values, load the knapsack with the most valuable collection it can hold

Search Problems: SEARCH RANDOM

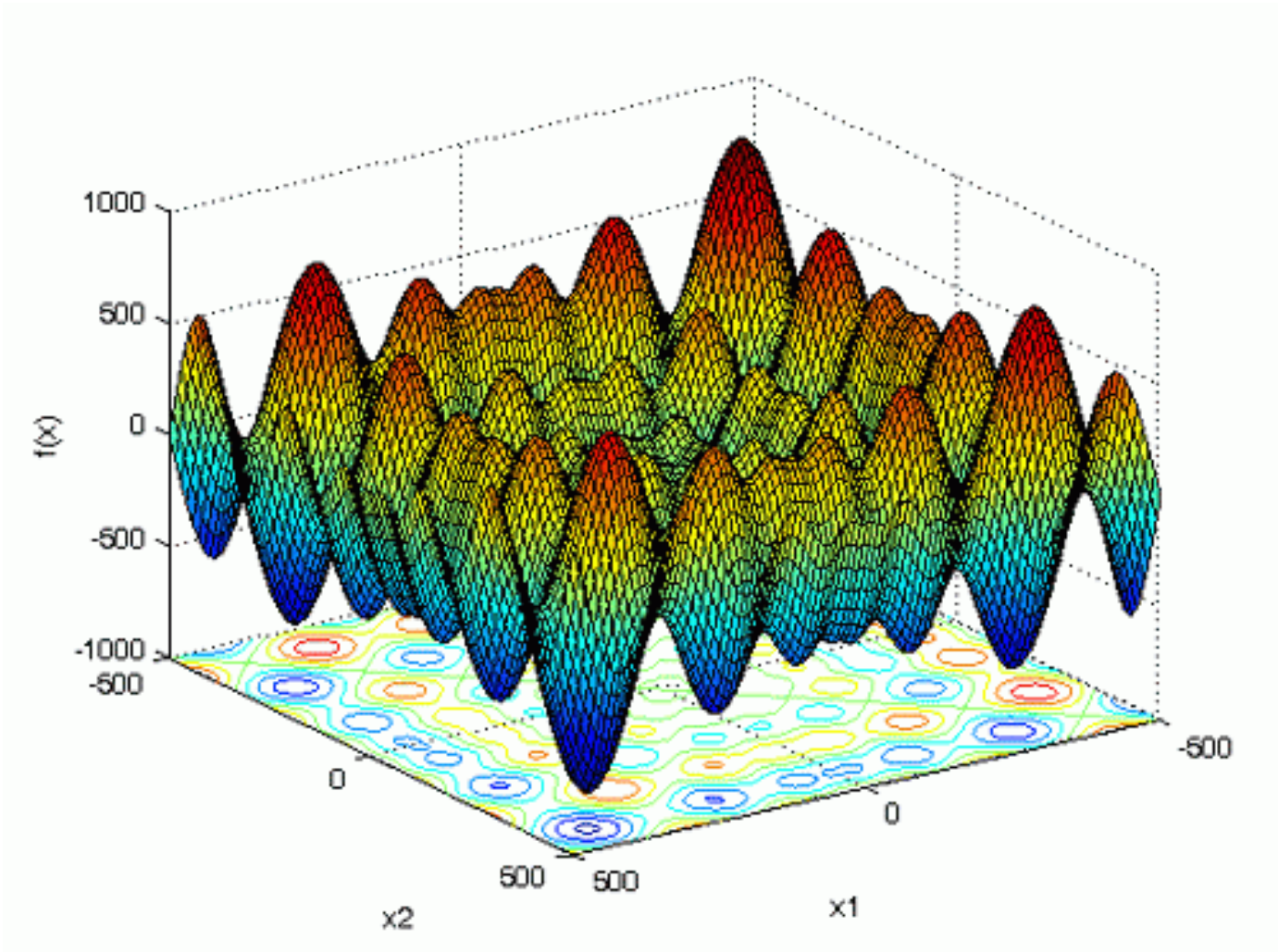
An important problem of optimization seeks the maximum value of a function $F(X_1, X_2, ..., X_N)$ for which a formula is given.

Many solution methods begin by picking a random starting value, then sampling the function "nearby" and moving in the direction where the function increases. More such steps are taken until no further increase is found.

Since there may be many local peaks that are not the highest one, it can be useful to carry out this process using many randomly chosen starting points.

Search Problems: SEARCH RANDOM

If you really need to find the highest hill, drop a lot of paratroopers at random, and tell them to move uphill.



Linear Algebra Problems

Linear algebra problems are

- **scalable**, you can make test problems of any size
- **measurable**, you can write down in advance the work needed,
- **realistic**, they occur in very many computational settings
- **unrealistic**, your own computations probably won't be so logic-free, computation-intensive, regular memory access

Linear Algebra Problems

- **DOT**, scalar dot product of two vectors
- **SAXPY**, add multiple of one vector to another
- **MV**, Matrix * Vector
- **MM**, Matrix * Matrix
- **JACOBI**, solve sparse A*x=y
- **SGEFA**, solve dense A*x=y

Linear Algebra Problems: DOT, SAXPY, MV, MM

The DOT, MV and MM tasks are very clean, very busy linear algebra operations.

They are easy to analyze and program.

But even these "almost perfectly straightforward" tasks can be programmed well or poorly!

1. **DOT**: $s = x * y$, (scalar result)
2. **SAXPY**: $z = s * x + y$, (vector result)
3. **MV** : $y = A * x$, (vector result)
4. **MM**: $C = A * B$, (matrix result)

Linear Algebra Problems: DOT, SAXPY, MV, MM

For vectors of length N, and matrices of order N by N, the FLOP count is

- **DOT** N
- **SAXPY** N
- **MV** $N * N$
- **MM** $N * N * N$

Linear Algebra Problems: DOT, SAXPY, MV, MM

These linear algebra tasks are so mathematically simple that it's hard to see how a performance issue could arise.

But as an illustration, consider the two implementations of the **MV** task.

For a 1000x1000 matrix, the (I,J) order runs at 17 MegaFLOPs, the (J,I) order at 40 MegaFLOPS.

Even on a sequential machine, there are "communication" problems!

Linear Algebra Problems: MV in the I,J order

```
integer i, j, n
real a(n,n), x(n),  y(n)

do i = 1, n
  y(i) = 0.0
  do j = 1, n
    y(i) = y(i) + a(i,j) * x(j)
  end do
end do
```

Linear Algebra Problems: MV in the J,I order

```
integer i, j, n
real a(n,n), x(n),  y(n)

do i = 1, n
  y(i) = 0.0
end do

do j = 1, n
  do i = 1, n
    y(i) = y(i) + a(i,j) * x(j)
  end do
end do
```

Linear Algebra Problems: JACOBI

The **JACOBI** task is a sort of inverse of the **MV** task.

Given matrix A and vector y, find x so that $y = A * x$.

We'll assume A allows the use of *Jacobi iteration*.

Linear Algebra Problems: JACOBI

Jacobi iteration improves an estimated solution x.

1. In equation 1, use old values for $x(2)$ through $x(n)$, and solve for new $x(1)$;
2. In equation 2, use old values for $x(1)$, and $x(3)$ through $x(n)$, and solve for new $x(2)$;
3. *...and so on until...*
4. In equation n, use old values for $x(1)$ through $x(n-1)$, and solve for new $x(n)$

At the end of this process (and only then!) replace all the old values by the new values.

The updates can happen independently, and in any order.

Linear Algebra Problems: SGEFA

The **SGEFA** task, like the **JAC task**, seeks x so that $y = A * x$.

The all-purpose solution method is *Gauss elimination*.

1. Eliminate $x(1)$ from equations 2 through n;
2. eliminate $x(2)$ from equations 3 through n.
3. *...and so on until...*
4. eliminate $x(n-1)$ from equation n.
5. solve equation n for $x(n)$
6. solve equation $n-1$ for $x(n-1)$
7. *...and so on until...*
8. solve equation 1 for $x(1)$.

Linear Algebra Problems: SGEFA

SGEFA searches through the data for maximum pivot values, moves data to put the pivot row in a special place, and does a lot of arithmetic, making it a "typical" scientific calculation.

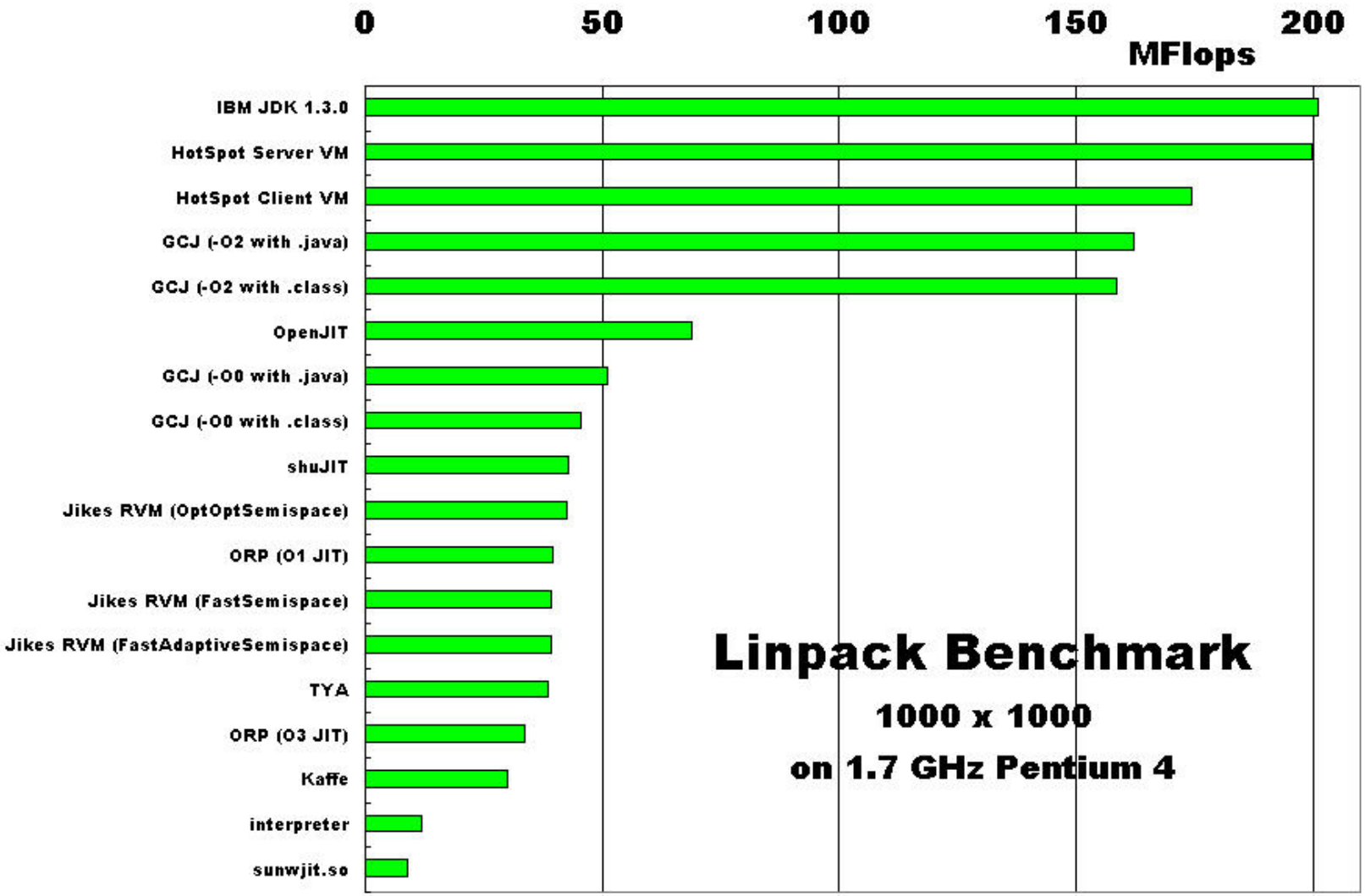
For an N by N matrix, the FLOP count is

- $2N^3/3$ for the elimination,
- N^2 for the back substitution;

The case $N = 100$ was the original **LINPACK Benchmark**. Soon $N = 1000$ gave more reliable results. The current distributed memory version, **HPL**, lets N increase to suit the system.

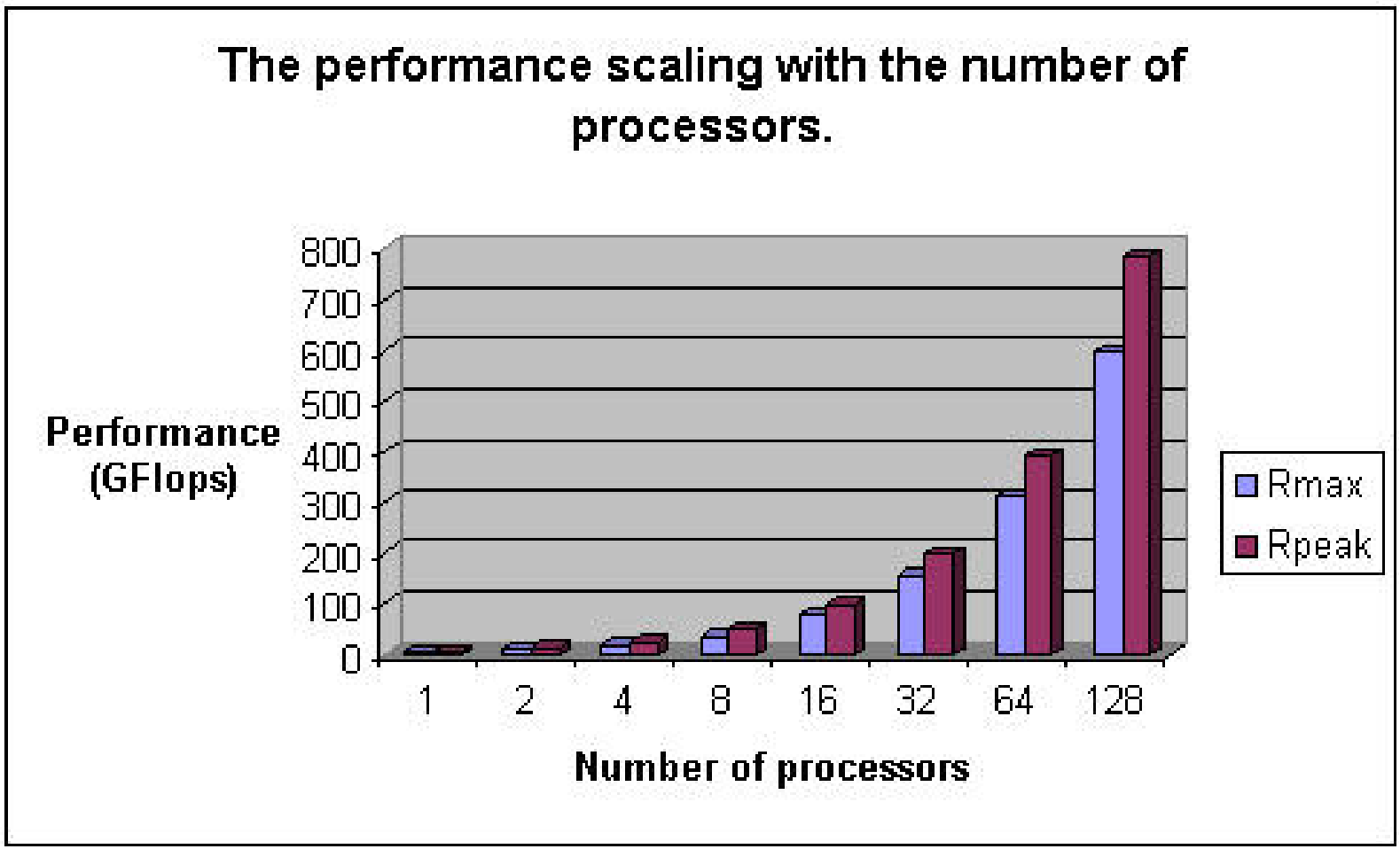
Linear Algebra Problems: SGEFA

The Linpack Benchmark can compare language implementations and compiler options (here, JAVA):



Linear Algebra Problems: SGEFA

The Linpack Benchmark can be used to examine parallel speedup



Model Programming Tasks

In our discussions and exercises, I will refer to some of the tasks we have mentioned.

Many scientific calculations have parts that are understandable in terms of these model tasks.

Understanding these simple tasks will help you to identify corresponding features of your own programs, and thus help you apply parallel programming ideas.