Data parallel algorithms[1]

(Guy Steele): The data-parallel programming style is an approach to organizing programs suitable for execution on massively parallel computers.

In this lecture, we will—

- characterize the _____ programming style,
- examine the building blocks used to construct data-parallel programs, and
- see how to fit these building blocks together to make useful algorithms.

All programs consist of code and data put together. If you have more than one processor, there are various ways to organize parallelism.

- Control parallelism: Emphasis is on extracting parallelism by orienting the program's organization around the parallelism in the code.
- _____ parallelism: Emphasis is on organizing programs to extract parallelism from the organization of the data.

With data parallelism, typically all the processors are at roughly the same point in the program.

Control and data parallelism vs. SIMD and MIMD.

- You may write a data-parallel program for a MIMD computer, or
- a control-parallel program which is executed on a SIMD computer.

Emphasis in this talk will be on styles of organizing programs. It becomes an engineering issue whether it is appropriate to organize the hardware to match the program.

The sequential programming style, typified by C and Pascal, has building blocks like—

- scalar arithmetic operators,
- control structures like **if** … **then** … **else**, and
- subscripted array references.

[1]Video © 1991, Thinking Machines Corporation. This video is available from University Video Communications, http://www.uvc.com.

The programmer knows essentially how much these operations cost. E.g., addition and subtraction have similar costs; multiplication may be more expensive.

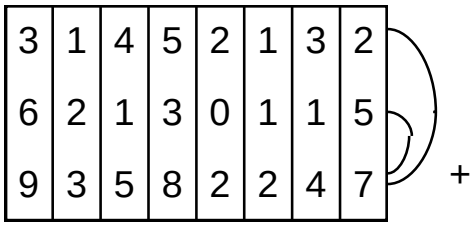
To write data-parallel programs effectively, we need to understand the cost of data-parallel operations.

- Elementwise operations (carried on independently by processors; typically _____ operations and tests).
- Conditional operations (also elementwise, but some processors may not participate, or act in various ways).
- Replication
- _____
- Permutation
- Parallel prefix (scan)

An example of an elementwise operation:

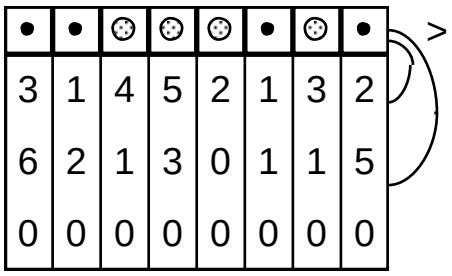
Elementwise addition

$C = A + B$



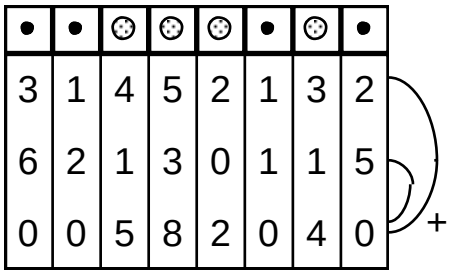
Elementwise test

if ($A > B$)



The results can be used to “conditionalize” future operations:

if ($A > B$) $C = A + B$

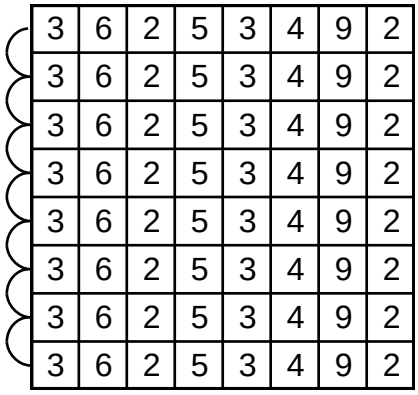


The set of bits that is used to conditionalize the operations is frequently called a *condition mask* or a *context*. Each processor can perform different computations based on the data it contains.

Building blocks

Communications operations:

- _____: Get a single value out to all processors. This operation happens so frequently that is worthwhile to support in hardware. It is not unusual to see a hardware bus of some kind.
- Spreading (nearest-neighbor grid). One way is to have each row copied to its nearest neighbor.

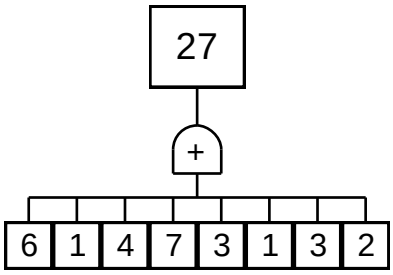


A better way is to use a copy-scan:

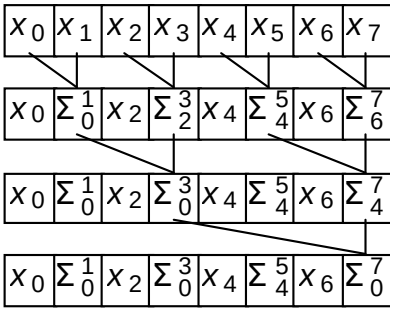
- On the first step, the data is copied to the row that is directly below.
- On the second step, data is copied from each row that has the data to the row that is two rows below.
- On the third step, data is copied from each row to the row that is four rows below.

In this way, the row can be copied in logarithmic time, if we have the necessary interconnections.

- _____—essentially the inverse of broadcasting. Each processor has an element, and you are trying to combine them in some way to produce a single result.



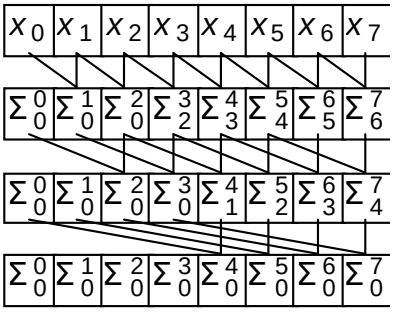
Summing a vector in logarithmic time:



Most of the time during the course of this algorithm, most processors have *not* been busy.

So while it is fast, we haven't made use of all the processors.

Suppose you don't turn off processors; what do you get? Vector sum-prefix (sum-scan).

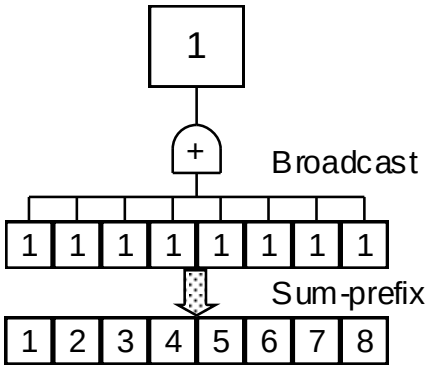


Each processor has received the sum of what it contained, plus all the processors preceding it.

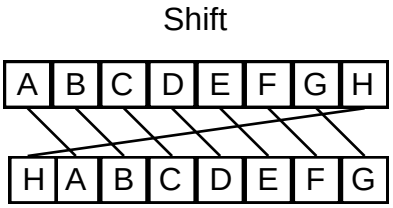
We have computed the sums of all *prefixes*—initial segments—of the array.

This can be called the checkbook operation; if the numbers are a set of credits and debits, then the prefixes are the set of running balances that should appear in your checkbook.

- _____. We wish to assign a different number to each processor.

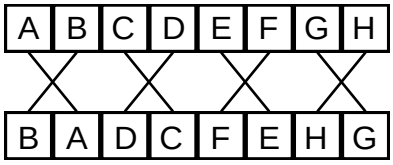


- Regular permutation.

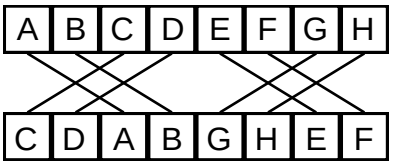


Of course, one can do shifting on two-dimensional arrays too; you might shift it one position to the north.

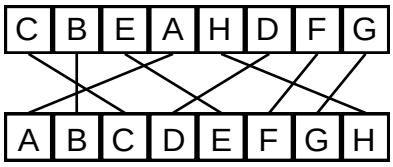
Another kind of permutation is an odd-even swap:



Distance 2^k swap:



Some algorithms call for performing irregular permutations on the data.



The permutation depends on the data. Here we have performed a sort. (Real sorting algorithms have a number of intermediate steps.)

Example: image processing

Suppose we have a rocket ship and need to figure out where it is.

Some of the operations are strictly local. We might focus in on a particular region, and have each processor look at its values and those of its neighbor.

This is a local operation; we shift the data back and forth and have each processor determine whether it is on a boundary.

When we assemble this data and put it into a global object, the communication patterns are dependent on the data; it depends on where the object happened to be in the image.

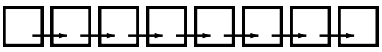
Irregularly organized data

Most of our operations so far were on arrays, regularly organized data.

We may also have operations where the data are connected by pointers.

In this diagram, imagine the processors as being in completely different parts of the machine, known to each other only by an address.

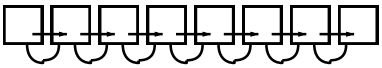
_____ doubling:



I originally thought that nothing could be more essentially sequential than processing a linked list. You just can't find the third one without going through the second one. But I forgot that there is processing power at each node.

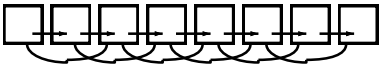
The most important technique is *pointer doubling*. This is the pointer analogue of the spreading operation we looked at earlier to make a copy of a vector into a matrix in a logarithmic number of steps.

In the first step, each processor makes a copy of the pointer it has to its neighbor.



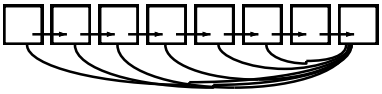
In the rest of the steps, each processor looks at the processor it is pointing to with its extra pointer, and gets a copy of *its* pointer.

In the first step, each processor has a pointer to the next processor. But in the next step, each processor has a pointer to the processor two steps away in the linked list.

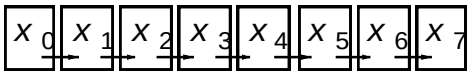


In the next step, each processor has a pointer to the pointer four processors away (except that if you fall off the end of the chain, you don't update the pointer).

Eventually, in a logarithmic number of steps, each processor has a pointer to the end of the chain.

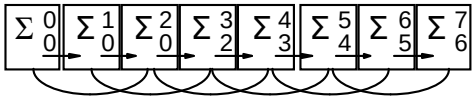


How can this be used? In partial sums of a linked list.

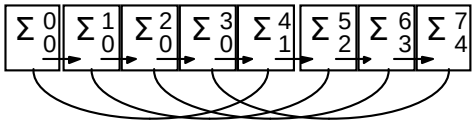


At the first step, each processor takes the pointer to its neighbor.

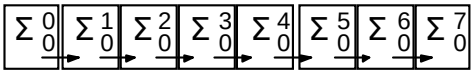
At the next step, each processor takes the value that it holds, and adds it into the value in the place pointed to:



Now we do this again:



And after the third step, you will find that each processor has gotten the sum of its own number plus all the preceding ones in the list.



Speed vs. efficiency: In sequential programming, these terms are considered to be synonymous. But this coincidence of terms comes about only because you have a single processor.

In the parallel case, you may be able to get it to go fast by doing extra work.

Let's take a look at the serial vs. parallel algorithm for summing an array.

_____-Reduction

	Serial	Parallel
Processors	1	N
Time steps	N–1	log N
Additions	N–1	N–1
Cost	N–1	N log N
Efficiency	1	$\frac{1}{\log N}$

Sum – Prefix

	Serial	Parallel
Processors	1	n
Time steps	n–1	log n
Additions	n–1	n (log n–1)
Cost	n–1	n log n
Efficiency	1	$\frac{\log n-1}{\log n}$

The serial version of sum–prefix is similar to the serial version of sum–reduction, but you save the partial sums. You don't need to do any more additions, though.

In the parallel version, the number of additions is much greater. You use n processors, and commit log n time steps, and nearly all of them were busy.

As n gets large, the efficiency is very close to 1. So this is a very efficient algorithm. But in some sense, the efficiency is bogus; we've kept the processors

busy doing more work than they had to do. Only $n-1$ additions are really required to compute sum–prefix. But $n(\log n-1)$ additions are required to do it fast.

Thus, the business of measuring the speed and efficiency of a parallel algorithm is tricky. The measures I used are a bit naïve. We need to develop better measures.

Exercise: Submit your answers here.

Calculate the speedup of summing a vector using copy-scan (turning off the processors that are not in use).

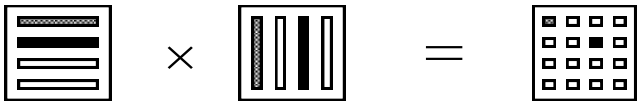
- How long does it take to sum the vector serially?
- How long does it take to sum it using copy-scan?
- What is the speedup?

What is the *efficiency* (speedup ÷ # of processors) of summing a vector with copy-scan?

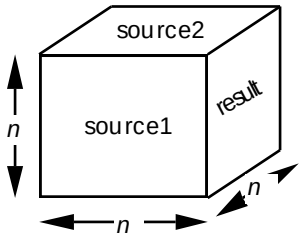
In the parallel version of summing an array via sum-prefix, a "bogus" efficiency is mentioned. What would be the "non-bogus" efficiency of the same algorithm?

Putting the building blocks together

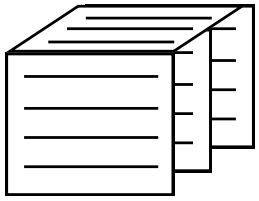
Let's consider *matrix multiply*.



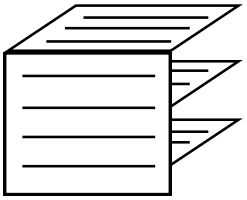
One way of doing this with a brute-force approach is to use n^3 processors.



1. Replicate. The first step is to make copies of the first source array, using a spread operation.

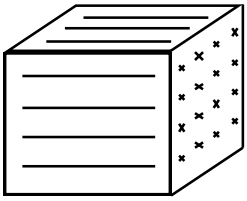


2. Replicate. Then we will do the same thing with the second source, spreading those down the cube.

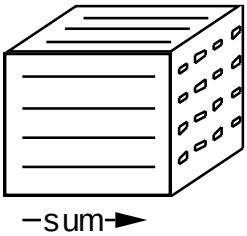


So far, we have used $O(\log n)$ time.

3. Elementwise multiply. n^3 operations are performed, one by each processor.



4. Perform a parallel sum operation, using the doubling-reduction method.



We have multiplied two matrices in $O(\log n)$ time, but at the cost of using n^3 processors.

Brute force: n^3 processors $O(\log n)$ time

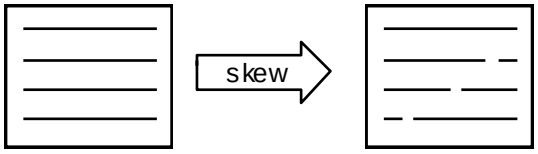
Also, if we wanted to add the sum to one of the matrices, it's in the wrong place, and we would incur an additional cost to move it.

Cannon's method

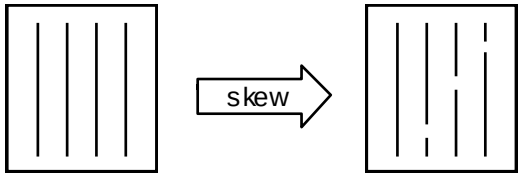
There's another method that only requires n^2 processors. We take the two source arrays and put them in the same n^2 processors. The result will also show up in the same n^2 processors.

We will pre-_____ the two source arrays.

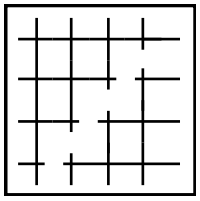
- The first array has its rows skewed by different amounts.



- The columns of the second array are skewed.

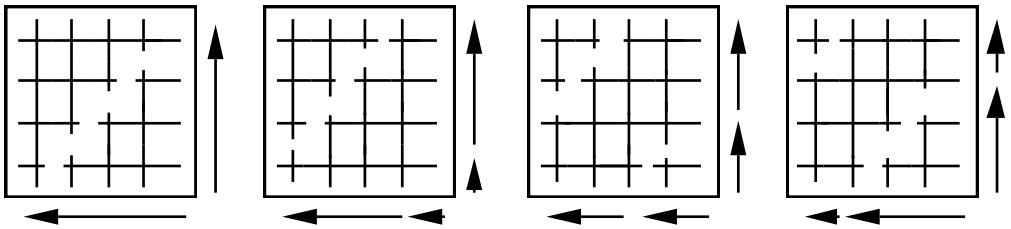


The two arrays are overlaid, and they then look like this:



This is a systolic algorithm; it rotates both of the source matrices at the same time.

- The first source matrix is rotated horizontally.
- The second source matrix is rotated vertically.



At the first time step, the 2nd element of the first row and the 2nd element of the first column meet in the upper left corner. They are then multiplied and accumulated.

At the second time step, the 3rd element of the first row and the 3rd element of the first column meet in the upper left corner. They are then multiplied and accumulated.

At the third time step, the 4th element of the first row and the 4th element of the first column meet in the upper left corner. They are then multiplied and accumulated.

At the fourth time step, the 1st element of the first row and the 1st element of the first column meet in the upper left corner. They are then multiplied and accumulated.

The same thing is going on at all the other points of the matrix.

The _____ serves to cause the correct elements of each row and column to meet at the right time.

Cannon's method: n^2 processors $O(n)$ time

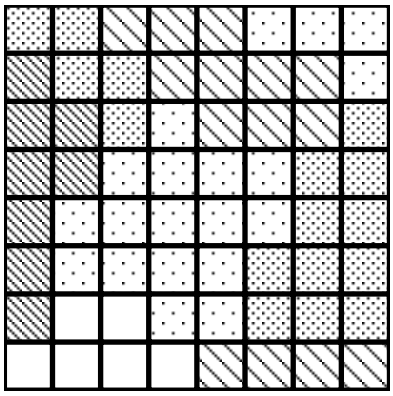
An additional benefit is that the matrix ends up in the right place.

Labeling regions in an image

Let's consider a really big example.

Instead of the rocket ship earlier in the lecture, we'll consider a smaller region. (This is one of the problems in talking about data-parallel algorithms. They're useful for really large amounts of data, but it's difficult to show that on the screen.)

We have a number of regions in this image. There's a large central "green" region, and a "red-orange" region in the upper right-hand corner. Some disjoint regions have the same color.



We would like to compute a result in which each region gets assigned a distinct number.

We don't care which number gets assigned, as long as the numbers are distinct (even for regions of the same color.

0	0	2	2	2	5	5	5
8	0	0	2	2	2	2	5
8	8	0	19	2	2	2	23
8	8	19	19	19	19	23	23
8	19	19	19	19	19	23	23
8	19	19	19	19	23	23	23
8	49	49	19	19	23	23	23
49	49	49	49	60	60	60	60

For example, here the central green region has had all its pixels assigned the value 19.

The squiggly region in the upper left corner has received 0 in all its pixels.

The region in the upper right, even though the same color as the central green region, has received a different value.

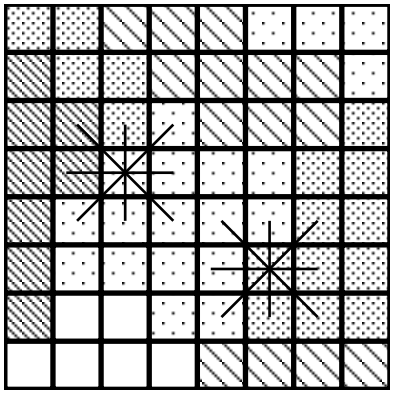
Let's see how all the building blocks we have discussed can fit together to make an interesting algorithm.

First, let's assign each processor a different number.

Here I've assigned the numbers sequentially across the rows, but any distinct numbering would do.

We've seen how the enumeration technique can do this in a logarithmic number of time steps.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63



Next, we have each of the pixels examine the values of its eight neighbors.

This is easily accomplished using regular _____— namely, shifts of the matrix.

We shift it up, down, left, right, to the northeast, northwest, southeast, and southwest.

This is enough for each processor to do elementwise computation and decide whether it is on the border.

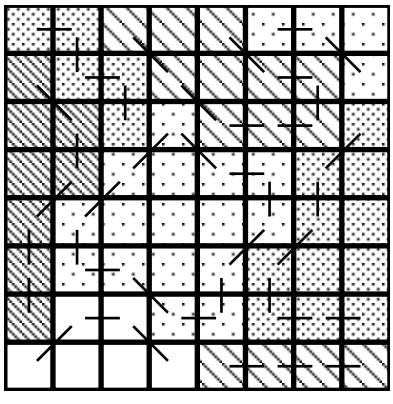
(There are messy details, but we won't discuss them here, since they have little to do with parallelism.)

The next computation will be carried out only by processors that are on the borders (an example of conditional operation).

We have each of the processors again consider the pixel values that came from its neighbors, and

inquire again, using shifting, if each of its neighbors are border elements.

This is enough information to figure out which of your neighbors are border elements in the same region, so you can construct pointers to them.



0	1	2		4	5	6	
8	9	10	11		13	14	15
	17	18	19	20	21	22	23
	25	26		28	29	30	
32	33				37	38	
40	41	42		44	45		
48	49	50	51	52	53	54	55
56			59	60	61	62	63

Now we have stitched together the borders in a linked list.

We now use the pointer-doubling algorithm. Each pixel on the borders considers the number that it was assigned in the enumeration step.

We use the pointer-doubling algorithm to do a reduction step using the **min** operation.

0	0	2		2	5	5	
8	0	0	2	2	2	2	5
	8	0	19	2	2	2	23
	8	19		19	19	23	
8	19				19	23	
8	19	19		19	23		
8	49	49	19	19	23	23	23
49			49	60	60	60	60

Each linked list performs pointer-doubling around that list, and determines which number is the smallest in the list.

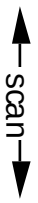
Then another pointer-doubling algorithm makes copies of that minimum all around the list.

Finally, we can use _____ operation, not on linked lists, but by operating on the columns (or the rows) to copy the processor labels from the borders to the rows.

Other items, particularly those on the edge, may need the numbers propagated up instead of down. So you do a scan in both directions.

The operation used is a non-commutative operation that copies the old number from the neighbor, unless it comes across a new number.

0	0	2	2	2	5	5	5
8	0	0	2	2	2	2	5
8	8	0	19	2	2	2	23
8	8	19	19	19	19	23	23
8	19	19	19	19	19	23	23
8	19	19	19	19	23	23	23
8	49	49	19	19	23	23	23
49	49	49	49	60	60	60	60



This is known as Lim's algorithm.

Region labeling: $O(n^2)$ processors. $O(\log n)$ time

(Each of the steps was either constant time or $O(\log n)$ time.)

Data-parallel programming makes it easy to organize operations on large quantities of data in massively parallel computers.

It differs from sequential programming in that its emphasis is on operations on entire sets of data instead of one element at a time.

You typically find fewer loops, and fewer array subscripts.

On the other hand, data-parallel programs are like sequential programs, in that they have a single thread of control.

In order to write good data-parallel programs, we must become familiar with the necessary building blocks for the construction of data-parallel algorithms.

With one processor per element, there are a lot of interesting operations which can be performed in constant time, and other operations which take logarithmic time, or perhaps a linear amount of time.

This also depends on the connections between the processors. If the hardware doesn't support sufficient connectivity among the processors, a communication operation may take more time than would otherwise be required.

Once you become familiar with the building blocks, writing a data-parallel program is just as easy (and just as hard) as writing a sequential program. And, with suitable hardware, your programs may run much faster.

Exercise: Run through Lim's algorithm on the grid given here.

Questions and answers: [not shown during class] *Question:* (Bert Halstead): Do you ever get into problems when you have highly data-dependent computations, and it's hard to keep more than a small fraction of the processors doing the same operation at the same time?

Answer: Yes. That's one reason for making the distinction between the data-parallel style and _____ hardware. The best way to design a system to give you the most flexibility without making it overly difficult to control is, I think, still an open research question.

Question (Franklin Turback): Your algorithms seem to be based on the assumption that you actually have enough processors to match the size of your problem. If you have more data than processors, it seems that the logarithmic time growth is no longer justified.

Answer: There's no such thing as a free lunch. Making the problem bigger makes it run slower. If you have a much larger problem that won't fit, you're going to have to buy a larger computer.

Question: How about portability of programs to different machines?

Answer: Right now it's very difficult, because so far, we haven't agreed on standards for the right building blocks to support. Some architectures support some building blocks but not others. This is why you end up with non-portabilities of efficiencies of running times.

Question: For dealing with large sparse matrices, there are methods that we use to reduce complexity. If this is true, how do you justify the overhead cost of parallel processing?

Answer: Yes, that is true. It would not be appropriate to use that kind of algorithm on a sparse matrix, just as you don't use the usual sequential triply-nested loop.

_____ processing on a data-parallel computer calls for very different approaches. They typically call for the irregular communication and permutation techniques that I illustrated.

Question: What about non-linear programming and algorithms like branch-and-bound?

Answer: It is sometimes possible to use data-parallel algorithms to do seemingly unstructured searches, as on a game tree, by maintaining a work queue, like you might do in a more control-parallel, and at every step, taking a large number of task items off the queue by using an enumeration step and using the results of that enumeration to assign them to the processors.

This may depend on whether the rest of the work to be done is sufficiently similar. If it's not, then control parallelism may be more appropriate.

Question: With the current software expertise in 4GLs for sequential machines, do you think that developing data-parallel programming languages will end up at least at 4GL level?

Answer: I think we are now at the point where we know how to design data-parallel languages at about the level of expressiveness as C, Fortran, and possibly Lisp. I think it will take awhile before we can raise our level of understanding to the level we need to design 4GLs.