Parallel Computing Models

Tom Robey and Bob Robey

- Why parallel? - A history
- Parallel Strategies
- Scalability
- Performance Model

Supercomputing Challenge Kickoff 2009-2010
October 25-26, 2009

Nothing Doubles Forever

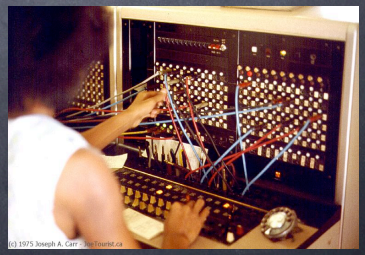
- Computers emphasized faster clock speeds (Dr. Dobbs, January 1994)
  - Memory capacity doubles every 1.5 years
  - CPU performance doubles every 2 years
  - Data-bus width doubles every 5 years
  - DRAM chip speed doubles every 7 years
- But clock speeds are no longer the driver of performance gains

Switch to Parallel Computing

- Move from vectorized Cray supercomputers to massively parallel machines
- Move from faster PC clock speeds to multi-core
- Cheaper to make many processors than a single very fast processor

Parallel Hardware

- Cheap processors
- Expensive low-latency communication
  - Wiring to all other processors does not scale
  - Early work focused on efficient mesh topology (ring, tree, hypercube, etc.)
  - Tunneling technology made communication mesh irrelevant (mesh exercise)
  - Dynamic topology



Memory Topology

- Three primary memory architectures
  - shared memory
  - distributed memory
  - memory hierarchies (cache, memory, disk)

	Access Speed	Scalable	Synchronization	Bottleneck
Shared	Fast	No	Memory Access	Memory
Distributed	Slow	Yes	Message Passing	Network

Modern Architectures

- Multi-core nodes (shared memory)
- Specialized functions (CPU and GPU)
- Hierarchical memory (cache, memory, disk)
- Distributed memory for nodes

What is the Parallel Paradigm? (Parallel Software)

- Decompose the task into smaller tasks
- Assign the smaller tasks to processors to work on simultaneously
- Coordinate work and communicate when necessary
- Not all problems have the same amount of parallelism
- Solving problems on a parallel machine requires that we consider new approaches to programming

Computer Language Philosophies

- Fortran
  - Single purpose code with a moderate amount of development time
  - Relies on the compiler to optimize the code
- C/C++
  - Complex, multi-purpose code with a high amount of development time
  - Optimizing code is up to the programmers

Parallel Strategies

- Data Parallel (High Performance Fortran, SplitC) - split the data amongst the processors and let the compiler handle the communication
- Message Passing (MPI, PVM) - the programmer handles the communication
- Object-Oriented - distribute objects
- Task Parallelism - GPU
- Distributed Computing - CORBA, Web Services

Team Strategies

Your supercomputing team is much like a parallel computer, dividing up the work and hoping to accomplish more than a single person in a limited time frame.

- Think about the types of parallel strategies listed on the previous slide. Which strategies is your supercomputing team employing?
- Which tasks are inherently serial?
- What communication is required by your team's strategy?

Scalability

- **Amdahl's Law** (Strong Scaling)- Limits of scaling for a fixed size problem

$$S_A(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- **Gustafson's Law** (Weak Scaling) - Increase the amount of work with the number of processors

$$S_G(p) = p - \alpha(p-1)$$

S - speedup
p - number of processors
α - serial fraction

Is Scalability Important?

- Scalability is about getting a result in less time
- Many parallel problems will not fit in the memory of a single processor
- A parallel program is not just a serial program that has been ported; parallel programs often can do more physics than a serial program
- For some applications, distributed computing provides convenience to the users even if it does not result in speedup

Asymptotic Notation (Big O notation)

- Used when we are only interested in the behavior of a function as the independent variables get large.
  - If f(x) is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
  - If f(x) has constants as part of terms, they are omitted.
  - Example: $f(x) = 6x^2 + 4x + 2 = O(x^2)$

Isoefficiency Analysis

- A generalization of Amdahl's Law and Gustafson's Law

n - size of an input
p - number of processors
W - sequential execution time of the best sequential algorithm
$T_0(W,p)$ - parallel overhead such as communication time
T_p - parallel execution time using p processors

Start by calculating T_p. Then the parallel overhead is

$$T_0(W,p) = pT_p - W$$

If the **scalability condition**

$$T_0(W,p) = O(W)$$

can be met then algorithm on that architecture is **cost optimal**. The **isoefficiency function** is the equation

$$W = KT_0(W,p)$$

where K is a constant.

Communication Times

Communication times for a hypercube architecture. t_s is message start up time, t_w is 1/bandwidth, m is message size, and p is the number of processors.

Operation	Communication Time
One-to-all broadcast, All-to-one reduction	$\min((t_s + t_w) \log p, 2(t_s \log p + t_w m))$
All-to-all broadcast, All-to-all reduction	$t_s \log p + t_w m(p - 1)$
All-reduce	$\min((t_s + t_w m) \log p, 2(t_s \log p + t_w m))$
Scatter, Gather	$t_s \log p + t_w m(p - 1)$

Isoefficiency Example

Consider a problem of adding n numbers on p processors.

Non-cost optimal solution
Assuming p and n are powers of 2, then the parallel time is

$$T_p = O((n/p)\log p + n/p) = O((n/p)\log p)$$

where n/p is the local computation and (n/p)log p is the communication.

$$T_0(W,p) = pT_p - W = O(n\log p) - O(n) = O(n\log p)$$

The scalability condition cannot be met so the problem is not cost optimal.

Cost optimal solution

For this algorithm the parallel execution time is

$$T_p = O(n/p) + O(\log p) = O(n/p + \log p)$$

Then the overhead time is

$$T_0(W,p) = pT_p - W = O(p \log p)$$

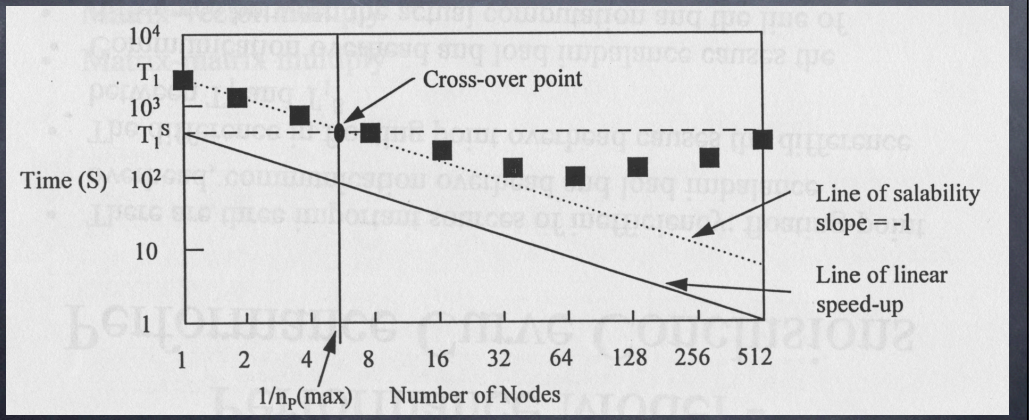
The scalability condition

$$O(p \log p) = O(n)$$

or that the parallel algorithm is cost optimal as long as $n = O(p \log p)$.

Performance Curve

- This curve assumes that the problem size does not change as we add nodes



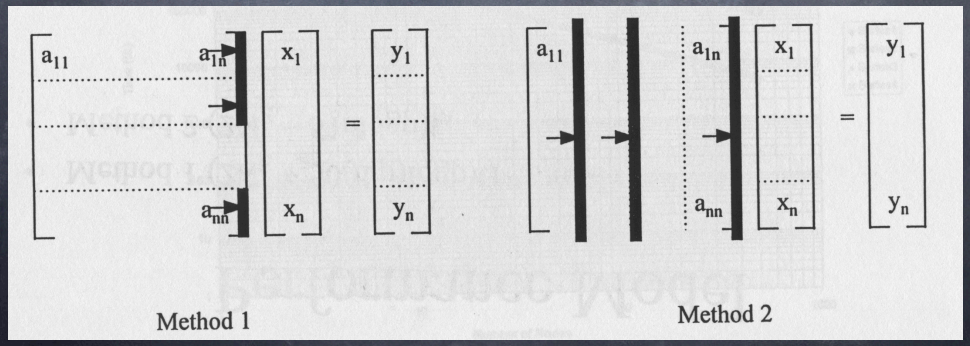
Matrix-Vector Multiply Introduction

- Matrix-vector multiply can either be thought of as
  - N vector-products of the rows of A with x
  - linear combination of the columns of A defined by x

$$\begin{bmatrix} a_{11} \dots a_{1n} \\ \dots \\ a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix} \qquad Ax=y$$

Matrix-Vector Multiply Introduction (Cont.)

- Method 1 - distribute blocks of rows of A and the entire x-vector to each processor
- Method 2 - distribute blocks of columns of A and blocks of the x-vector to each processor



Matrix-Vector Multiply Performance Model

Compute T_p for Method 1 and Method 2. Use the chart for communication times. The code in the following slide may help. What does the scalability condition say for each method?

Matrix-Vector Multiply MPI Code

```
 program main
 integer dim1, dim2, dim3
 parameter (dim1=80, dim2=10, dim3=dim1*dim2)
 include "mpif.h"
 integer ierr, rank, size, root
 real a(dim1, dim1), apart(dim3), ypart(dim1), y(dim1),
&      x(dim1),xpart(dim2)

 root = 0

 call MPI_INIT(ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

 write(*,*) `START process `, rank
```

```
if (rank .eq. root) then
   do j=1,dim1
     x(j) = 1.0
     do i=1,dim1
       a(i,j) = i+j
     enddo
   enddo
 endif
 call MPI_SCATTER(a, dim3, MPI_REAL, apart, dim3, MPI_REAL, root,
&                        MPI_COMM_WORLD, ierr)
 call MPI_SCATTER(x, dim2, MPI_REAL, xpart,dim2, MPI_REAL, root,
&                        MPI_COMM_WORLD, ierr)
 do i=1,dim2
   do j=1,dim1
     if (i .eq. 1) ypart(j) = 0.0
     ypart(j) = ypart(j) + xpart(i)*apart((i-1)*dim1+j)
   enddo
 enddo
 call MPI_REDUCE(ypart, y, dim1, MPI_REAL, MPI_SUM, root,
&                    MPI_COMM_WORLD, ierr)
 write (*,*) 'FINISH process ', rank
 call MPI_FINALIZE(ierr)
 end
```

Matrix-Vector Multiply HPF Code

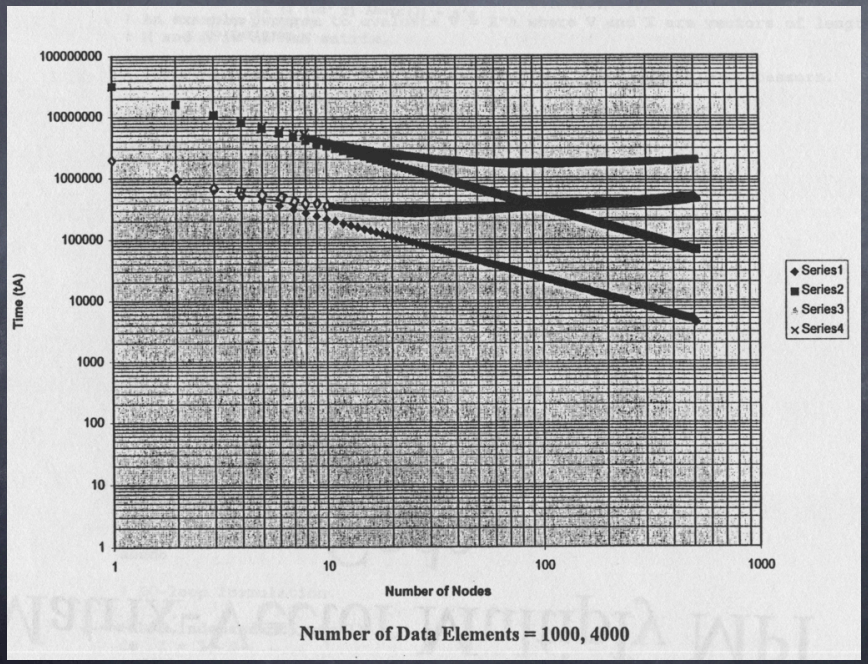
```
! An example program to evaluate V = X*A where V and X are vectors of length
! M and A is an MxN matrix

! Distribute array A by block columns.  Place X and V on all processors

implicit none
integer NPROCS
parameter (NPROCS = 3)
!HPF$ processors, dimension(NPROCS) :: PROCS

real A(M,N), X(M), V(N)
!HPF$ distribute (*,block) onto PROCS :: A
!HPF$ distribute (block) onto PROCS :: V

intrinsic dot_product, matmul
```

```
! Vector-matrix product using
!   1) Fortran 90 matmul formulation
!   2) Fortran 90 vector formulation
!   3) Fortran 90 element-wise formulation

! Matrix (matmul) formulation.
V = matmul(X,A)

! Vector formulation
!HPF$ independent, new(I)
do I = 1,N
  V(I) = dot_product(X, A(:,I))
enddo

! Do-loop formulation
!HPF$ independent, new(I)
do I=1,N
  V(I) = 0.0
  do J=1,M
    V(I) = X(J)*A(J,I) + V(I)
  enddo
enddo
```

Matrix-Vector Multiply HPF Comments

- Specify the number of processors with the HPF PROCESSORS directive
- Distribute the matrix A and the vector X in blocks over the processors. The vector V is on each processor
- The processors each have a block of columns and a block of the elements of size approximately N/NPROCS and compute the elements of V that are on each processor
- The alternative codes use a Fortran 90 vector notation or the traditional Fortran 77 (and 90) Do-loop notation
- The HPF INDEPENDENT directive is a hint to the HPF compiler that there are no loop iteration dependencies--the directive is not needed in the vector formulation as Fortran 90 states there are no dependencies by definition

Matrix-Vector Multiply Execution Plot



Matrix-Vector Multiply Conclusions

- Method 1 and Method 2 have approximately the same floating-point overhead
- Method 1 is superior to method 2 since it sends n/p instead of n data values

Conclusions

- Parallel programming is hard
- Knowing what you are trying to achieve (less time, bigger problem, more physics, it's cool, etc.) in a parallel program is an important start
- Different ways of organizing data and communications can have very different results
- It helps to have a performance model before creating a parallel program
- If the data does not agree with the performance model, why does it behave differently than expected?