Parallel Programming Languages

- Fortran 90/Matlab - a serial language with a parallel flavor,
- CM Fortran (CMF) - a parallel Matlab (or Fortran 90),
- HPF (High Performance Fortran) is intended to be a more portable and closely related language,
- Split-C is C augmented with a few constructs to make parallelism available and predictable.

Libraries for Parallel Machines

Parallel subroutines of certain common problems.

- Message passing subroutine libraries: PVM (Parallel Virtual Machine), MPI (Message Passing Interface).
- Active messages.
- LAPACK and ScaLAPACK - These are parallel numerical linear algebra libraries.

  solving systems of linear equations, least squares problems, and finding eigenvalues and eigenvectors.
- Multipol - a parallel distributed data structure library at Berkeley
- PETSc, LPARX, and KeLP and special purpose C++ libraries for the parallel solution of solving PDEs (partial differential equations), particle methods. scientific problems.
- RAPID, — run-time library for irregular problems.

Overview of architecture and programming models

Standard functions supported in any parallel architecture.

- 1. Parallelism, or getting processor to work simultaneously.
- 2.Interprocessor communication, or having processors exchange information.
- 3.Synchronization, such as getting processors to “agree” on the value of a variable, or a time at which to start or stop.

Parallelism, communication & Synchronization

in Computer architectures

- Parallelism

  The two main styles of parallelism are SIMD and MIMD, or single-instruction-multiple-data and multiple-instruction-multiple-data.
- Communication between processors.

  A simple cost model for sending n words = latency + n/bandwidth.
- Synchronization

  Synchronization refers to the need for two or more processors to agree on a time or a value of some data. The three most common types:
  - Mutual exclusion refers permitting just one processor access to a particular memory location at a time.

– Barriers involve each processor waiting at the same point in the program for all the others to "catch up", before proceeding.
– Memory consistency refers to a problem on shared memory machines with caches.

Problem sizes/performance terms

- 1 Mflop = 1 Megaflop = 10^6 floating point operations per second
- 1 Gflop = 1 Gigaflop =10^9 floating point operations per second
- 1 Tflop = 1 Teraflop =10^{12} floating point operations per second
- 1 MB = 1 Megabyte = 10^6 bytes
- 1 GB = 1 Gigabyte = 10^9 bytes
- 1 TB = 1 Terabyte = 10^{12} bytes
- 1 PB = 1 Petabyte = 10^{15} bytes

Grand Challenges or largest problems, like predicting the climate 50 years hence, are estimated to require computers computing at the rate of 1 Tflop, and with a memory size of 1 TB.

Performance of difference machines

Alpha and beta for various machines, where the message packet is taken to be 8 bytes.

They are are normalized so that the time for one flop done during double precision (8 byte) matrix multiplication is equal to 1. The speed of matrix multiplication in Mflops is also given.

Machine	alpha	beta	Matmul Mflops	Software
Alpha+Ether	38000	960	150	assumes PVM
Alpha+FDDI	38000	213	150	assumes PVM
Alpha+ATM1	38000	62	150	assumes PVM
Alpha+ATM2	38000	15	150	assumes PVM
HPAM+FDDI	300	13	20	
CM5	450	4	3	CMMD
CM5	96	4	3	Active Messages
CM5+VU	14000	103	90	CMMD

iPSC/860	5486	74	26	
Delta	4650	87	31	
Paragon	7800	9	39	
SP1	28000	50	40	
T3D	27000	9	150	Large messages(BLT)
T3D	100	9	150	read/write

Other results

"Message Passing Performance of Various Computers", by Jack Dongarra and Tom Dunigan, U. Tennessee CS report 95-299.

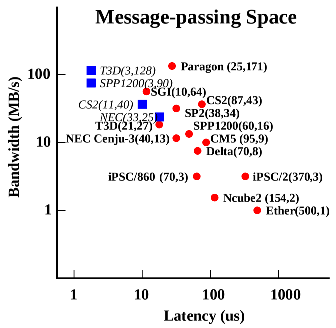

Figure 5: Latency/bandwidth space for 0-byte message (latency) and 1 MB message (bandwidth). Block points represent shared-memory copy performance.

Parallelism, communication & Synchronization in programming models

- Data parallelism:

  Applying the same operation, of set of operations, to all the elements of a data structure.

  Software: HPF.

- Message Passing:

  Running p independent sequential programs, written in sequential languages like C or Fortran, and communicating by calling subroutines like send(data,destination_proc) and receive(data,source_proc) for data communication.

  Software: PVM and MPI.

- Shared memory programming with threads.

  A natural programming model for shared memory machines. There

is a single program text, which initially starts executing on one processor. This program may execute a statment like "spawn(proc,0)", which will cause some other processor to execute subroutine proc with argument 0.

It can be extended for a distributed memory with a sofware support.

Software: Solaris threads. Split-C.

Measuring parallel performance

- T(p,n) is the time to solve a problem of size n on p processors. Sometimes we omit n and write T(p).
- T(1,n) or T(1) is the serial or sequential time, using the best serial algorithm.
- Speedup(p) = T(1)/T(p) measures how much faster you go on p processors than 1 processor. A "speedup plot" is Speedup(p) plotted versus p.
- Speedup(p)/p. An "efficiency plot" is Efficiency(p) plotted versus p. A good algorithm will have an efficiency near 1.

Notice that f you use a poor serial algorithm, T(1) and so Speedup(p) will be artificially large, and your parallel algorithm will look artificially good!

Shared Memory Programming with Multithreading

Multithreading is a programming style suitable for shared memory space MIMD machines, like the SGI Power Challenge, or the Sun SPARCcenter 2000.

- These machines have a single address space.
- Initially, a program consists of a single "thread" of control, as the main routine begins.
- The user controls parallelism by creating other "threads" of control, which are told to execute a subroutine of the users choice.
- The system schedules the threads to multiprocessors.
- Threads share the same address space and they may also synchronize with and wait for one another, e.g. when they write to the same data address.

SunOS Multi-thread architecture

Two "levels" of threads available in SunOS (Sun Operating System)

- User-level threads

  Scheduled entirely at the user level, so that the OS kernel does not have to get involved. This means they are relatively inexpensive to create, start, stop and synchronize, since anything involving the OS kernel is more expensive.
- LWP (light weight processes"). These are known to the kernel, and are consequently more expensive to create and use

User-level Multithreading on SUN

A thread is created by calling thr_create:

```
int thr_create( void *stack_addr,
                unsigned int stack_size,
                void (*func) (),
                void *arg,
                long flags,
                thread_id_t *new_thread )
```

- *stack_addr indicates where the thread's stack is to be based.
- stack_size indicate how large it may grow; defaults are available for both.
- When the thread starts running, it calls func(arg), which may be any routine the user wants. When func(arg) returns, the thread terminates.

- The flags describe how the thread will execute. For example, the thread may be suspended on creation, until a later thread_continue() call starts it. The thread may also either be allowed to "float", i.e. executed by any available physical processor, or tied down to execute on one of them;

thr_create returns a pointer to a thread identifier uniquely identifying the thread it creates; this can be used for synchronization purposes by other routines.

Example

```
thread_ptr[0].chunk = 0;
thread_ptr[0].tid = myID;
for (i = 1; i < NTHREADS; i++) {
      thread_ptr[i].chunk = i;
      if (thr_create(0, 0, move_fish,
                     (void*)&thread_ptr[i].chunk,
                     0, &thread_ptr[i].tid)) {
            perror("thr_create");
            exit(1);
      }
}
```


Synchronization between threads

- Thread_wait allows one thread to synchronize with another, by waiting for the thread specified by thread_id to exit:
  thread_id_t thread_wait(thread_id_t thread_id)
- Threads can signal one another by calling the following routines:
  int thread_stop(thread_id_t thread_id) int thread_continue(thread_id_t thread_id)
- Threads can synchronize using mutual exclusion (mutex), condition variables, semaphores, or reader/writers locks. For example

```
mutex_enter( &lock )
...
while ( not condition ) { cv_wait( &cond_var, &lock ) }
...
mutex_exit( &lock )
```


Why writing fast programs is hard

It is hard to get machine peak performance. Why?

- Program/data distribution – load balancing.
- Data locality: minimial communication overhead.
- Data locality: memory hierarchies.

Memory Hierarchy

- Registers. small, fast, and expensive
- Cache (perhaps more than one level).
- Main memory (perhaps local and remote).
- Disk. large, slow, and inexpensive

Designing fast linear algebra kernels

BLAS (Basic Linear Algebra Subroutines).

BLAS1 (Leve 1 BLAS) operate mostly on vectors (1D arrays):

- Saxpy/Daxpy operation: y = a*x+y.
  or For i=1 to n, y(i)=a*x(i)+y(i).
  S (single precision). D (double precision).
- Scaling computes y = a * x.
- Dot product computes $s = sum_{i=1}^{n} x(i) * y(i)$.

BLAS 2 operate mostly on a matrix (2D array) and a vector (or vectors):

- GEMV. Matrix-vector multiplication computes y = y + A*x.
- A rank-one update computes A = A + y*x'.
- A triangular solve solves y=T*x for x, where T is a triangular

matrix.

BLAS 3 BLAS3 operate on pairs or triples of matrices.

- GEMM. Matrix-matrix multiplication computes C = C + A*B.
- A multiple triangular solve solves Y = T*X for X, where T is a triangular matrix.

BLAS Performance

Any linear algebra algorithm can be expressed in terms of any of BLAS1, BLAS2, BLAS3.

The anwser is the performance. On IBM RS 6000/S90, peak performance 266Mflops.

- Small matrix/vector (n=100).
  BLAS3 (200 Mflops), BLAS2 (30 Mflops), BLAS1 (50 Mflops).
- Large matrix/vector (n=500).
  BLAS3 (250 Mflops), BLAS2 (100 Mflops), BLAS1 (65 Mflops).

Why BLAS3

Define:

- m = number of memory references to slow memory needed to read the input data, and write the output data back
- f = number of floating point operations
- q = f/m = average number of flops per slow memory reference.

Memory reference analysis:

- SAXPY $m = 3 * n, f = 2 * n, q = 2/3$.
  read each x(i), y(i) once, write y(i) once.
- SGEMV $m = n^2 + O(n), f = 2 * n^2, q = 2$.
  Read each A(i,j) once
- SGEMM $m = 4 * n^2, f = 2 * n^3, q = n/2$.
  read each A(i,j),B(i,j),C(i,j) once, write each C(i,j) once

Miss ratio

- Miss ratio — The fraction of all memory references (to fast or slow memory) that "miss" the fast memory, and need data from the slow memory.
- A good problem has a low miss ratio.
- Assume that each floating point operation involves 2 references to either fast or slow memory.

$$\mathit{Miss\ ratio} = \frac{\#\text{references to slow memory}}{\#\text{total memory references}}$$
$$= \frac{m}{2f} = \frac{1}{2q}.$$

Example

```
s = 0
for i = 1, n
     s = s + f(x(i))
end for
```

Assume:

- Each f(x(i)) takes $g-1$ operations, can be done in fast memory once x(i) is available.
- One can do at 1 Mflop on data in the fast memory, and fetching data from slow memory takes 10 times longer per word.
- x(i) is initially in slow memory,

Performance of this algorithm:

- t = 10*n + g*n microsecs
- 10*n microsecs to read each x(i), g*n to do g*n flops.
- Megaflop rate f/t = g/(g+10).
- As g increases, this megaflop rate approaches the top speed of 1 Mflop.

Example: Unblocked matrix multiply

```
for i=1 to n
    {Read row i of A into fast memory}
    for j=1 to n
       {Read C(i,j) into fast memory}
       {Read column j of B into fast memory}
       for k=1 to n
           C(i,j)=C(i,j) + A(i,k)*B(k,j)
       end for
       {Write C(i,j) back to slow memory}
    end for
end for
```

Assumptions:

- 1.There are just two levels in the hierarchy, fast and slow.
- 2.The small, fast memory has size M words, where $M << n^2$, but

$M \geq 4 * n$, so we can fit several whole rows or columns.

- 3.Each word is read from slow memory individually.
- 4.We have complete control over which words are transferred between the two levels.

Performance

m = # slow memory refs =

- n^3 read each column of B n times
- $+n^2$ read each row of A once for each i, and keep it in fast memory during the execution of the two inner loops
- $+2 * n^2$ read/write each entry of C once

$$Thus \quad m = n^3 + 3 * n^2$$

$$q = f/m = (2 * n^3)/(n^3 + 3 * n^2) \approx 2.$$

The same as GEMV.

Square blocked matrix multiply

Also called 2D blocked, or panel blocked.

Consider C to be an N-by-N matrix of n/N-by-n/N subblocks Cij, with A and B similarly partitioned.

```
for i=1 to N
   for j=1 to N
      {Read Cij into fast memory}
      for k=1 to N
         {Read Aik into fast memory}
         {Read Bkj into fast memory}
         Cij = Cij + Aik * Bkj
      end for
      {Write Cij back to slow memory}
   end for
end for
```


Assume that the fast memory is large enough so that the 3 subblocks Cij, Aik and Bkj fit in the fast memory; Thus.

$$M >= 3 * (n/N)^2, \ or \ N >= \sqrt{3/M} * n$$

Memory references $m = (2 * N + 2) * n^2$. Because

- $N * n^2$ read each Bkj N^3 times
- $+N * n^2$ read each Aik N^3 times
- $+2 * n^2$ read/write each Cij once

$$Thus \ \ q = f/m = (2 * n^3)/((2 * N + 2) * n^2) \approx n/N.$$

Increase q, and we can accelerate the algorithm. For the optimum, choose

$$N \approx \sqrt{3/M} * n, \ then \ \ q \approx \sqrt{M/3}.$$

LAPACK/ScaLPACK

LAPACK: libraries of linear algebra routines for solving standard linear algebra problems on shared memory machines, vector computers (Cray C90).

ScalPACK is for distributed memory machines.

Major issues:

- Reorganize an algorithm in either BLAS3 or BLAS2 operations to get the top performance.
- Layout dense matrices across multiple processor memories. tradeoffs among load balancing, minimizing communication, and exploiting the memory hierarchy.
- Design parallel software to be portable across a variety of architectures.

Notations

- A(i:j, k:l) refers to the submatrix of A in rows i through j and columns k through l. A(:,k:l) refers to columns k through l.
- A' is the transpose of A.
- A*B is the product of matrices A and B.
- sum(x) is the sum of the entries of x.
- norm(x) is the Euclidean norm of x : $\sqrt{sum(x.^2)}$

Basic problems of linear algebra

- Solve a system of linear equations Ax=b, where A is a general matrix.
- Solve a symmetric, positive definite (spd) system of linear equations Ax=b.
- Solve a least squares problem, i.e. find the x which minimizes norm(Ax-b).
- Find the eigenvalues λ_i and (possibly) eigenvectors x_i of A:

$$A * x_i = \lambda_i * x_i,$$

- Find the singular value decomposition (SVD) of A:

$$A = U * S * V'.$$

BALS based Gaussian Elimination

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i+1 to n
           A(j,k) = A(j,k) - m * A(i,k)
```

Store the multipliers m in the implicitly created zeros below the diagonal.

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
    for j = i+1 to n
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

BLAS2 GE

```
for i = 1 to n-1
   A(i+1:n, i) = A(i+1:n, i) / A(i,i)
   A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - A(i+1:n, i)*A(i, i+1:n)
```

One BLAS1 operation (scaling A(i+1:n,i)).

One BLAS2 operation (adding a rank-one matrix -A(i+1:n,i)*A(i,i+1:n), to A(i+1:n, i+1:n)).

LU factorization

The Gaussian Elimination code also produces A= L*U.

It can be used for solving A*x=b:

- 1) Factorize A = L*U.
- 2) Solve L*y = b for y by doing forward substitution.
- 3) Solve U*x = y for x by doing backward substitution.

Use BLAS3 in GE with partial pivoting

- Delayed updating.
- Process the matrix in blocks of b columns at a time, rather than one column at a time. b is called the block size.
- b is large enough to make matrix-matrix multiplication fast.

```
for ib = 1 to n-1 step b
 ... process matrix b columns at a time
   end = min(ib+b-1,n)
         ... points to end of current block of b columns

for i = ib to end
          ... LU factorize A(i:n,i:end) using BLAS2
    find and record k where |A(k,i)| = max_{i<=j<=n} |A(j,i)|
    if |A(k,i)|=0, exit with a warning that A is singular.
```



```
    if i != k, swap rows i and k of A
    A(i+1:n, i) = A(i+1:n, i) / A(i,i)
    A(i+1:n, i+1:end) = A(i+1:n, i+1:end)
                          - A(i+1:n, i)*A(i, i+1:end)
      ... only update columns i+1 to end

Let LL refer to the b-by-b lower triangular matrix
  whose subdiagonal entries are stored in A(ib:end,ib:end),
  and with 1s on the diagonal

A(ib:end, end+1:n) = LL \ A(ib:end, end+1:n)
   ... perform delayed update of A(ib_end, end+1:n) by
   ... solving n-end triangular systems of equations.

A(end+1:n, end+1:n) = A(end+1:n, end+1:n)
                       - A(end+1:n,ib:end)*A(ib(end,end+1:n)
      ... perform delayed update of rest of matrix using
      ... matrix-matrix multiplication
```