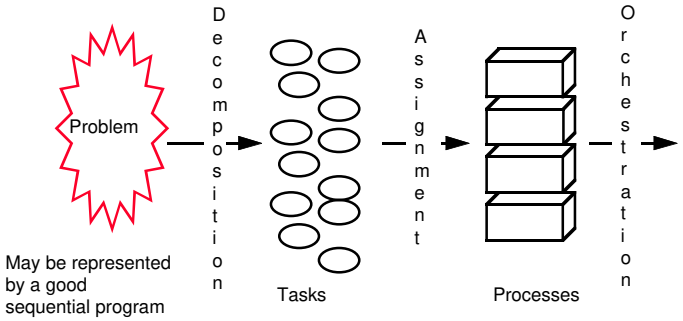
Parallel Software Basics

- Goals:
  - **understand what parallel programs "look like"**
    - » **shared memory and message passing programming models**
    - » **concrete examples**
  - **key issues and programming techniques for obtaining performance on parallel machines**
    - » **hardware / software interactions**
    - » **common issues and performance bottlenecks**
      - **where can we architects help? where not?**
  - **methodology for workload-driven architectural evaluation**
    - » **beyond speedup - scaling and working sets**

© Culler 1995

Steps in creating a parallel program

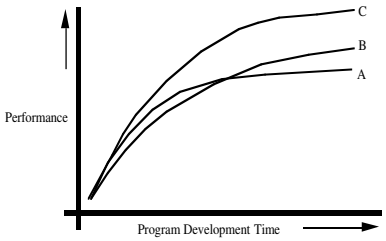
- Identifying Concurrency
- Decompose problem into constituent tasks
  - **chunks of work that constitute the overall computation**
- Define Assignment of work (and data) to processes
  - **abstract entity that performs tasks (thread)**
- Orchestrate dependences



© Culler 1995

Goal: sufficiency, not optimality

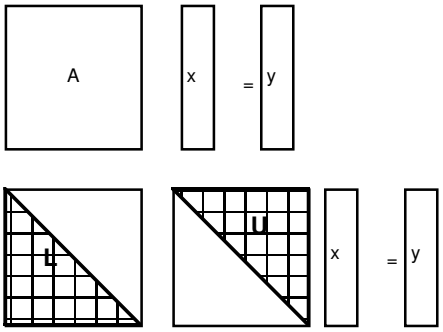
- Programming perspective
  - **high performance**
  - **low resource usage**
  - **low development time**
- Architecture perspective
  - **high performance**
  - **low hardware complexity**
  - **low development time**



(handware complexity)

© Culler 1995

Example: LU Decomposition



Simplification of the core of LINPACK benchmark.

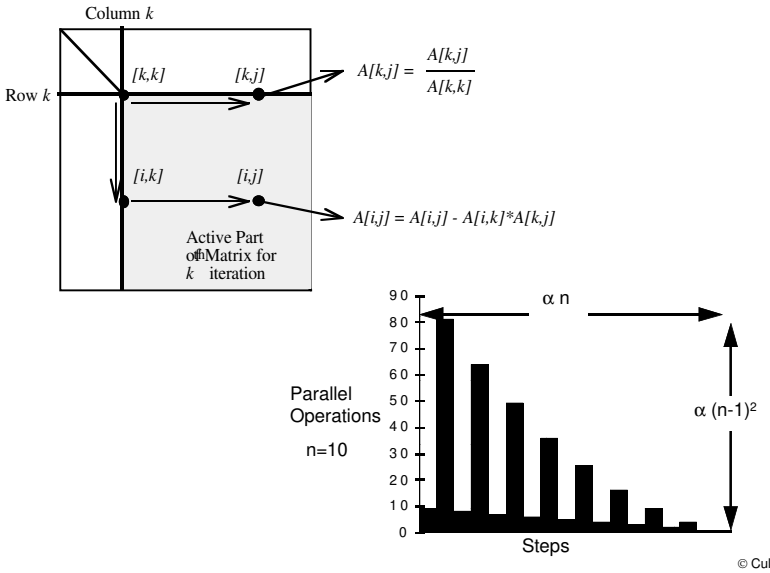
© Culler 1995

Sequential Program (by rows)

```
int n; double **A;
main()
begin
    read(n);                          /* read input parameters */
    A <– malloc (a 2-d array of size n by n doubles);
    initialize(A);                    /* initialize the matrix A somehow */
    LU(A);                              /* call the routine to factor A */
end main
procedure LU (A)                      /* LU factor the matrix A */
    double A(n,n);
begin
    for k <– 0 to n-1 do              /* loop over all diagonal elements */
      for j <– k+1 to n-1 do          /*  scale pivot row by diagonal element*/
        A[k,j] <– A[k,j] / A[k,k];
      for i <– k+1 to n-1 do          /* for all elements in the column */
        for j <– k+1 to n-1 do        /* for all elements in the corr. row */
          A[i,j] <– A[i,j] - A[i,k]*A[k,j];
    endfor
```

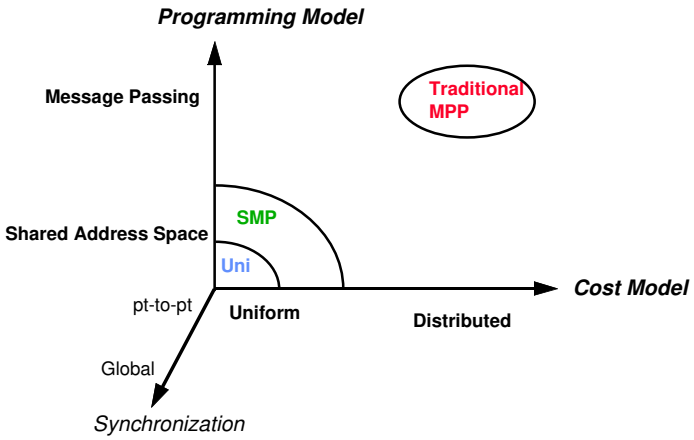

© Culler 1995

Identify Concurrency



© Culler 1995

Programming Approaches



© Culler 1995

SPMD Shared Address

```
int n, nprocs;                  /* matrix dimension and number of processors to be used
double **A;                     /* Global Data structure
BARDEC (bar1);                  /* barrier declaration for global synchronization between steps

main()
begin
   read(n);       read(nprocs);    /* read input matrix size and number of processes
   A <– G_MALLOC (a two-dimensional array of size n by n doubles);
                                   /* create processes
   CREATE (nprocs-1 processes that start executing LU(A));
   LU(A);                          /* main process becomes a worker too
   WAIT_FOR_END;                   /* wait for all child processes created to terminate
end main

procedure LU( A[n,n] )
    . . .

begin
```

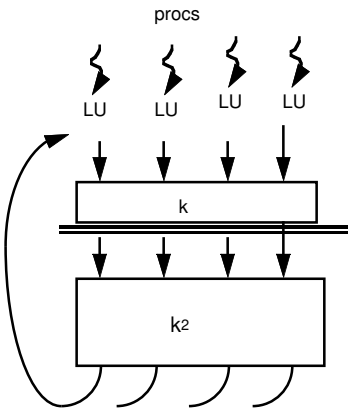
Automatic in many programming environments

© Culler 1995

Shared Address, uniform, global

```
procedure LU( A[n,n] )
begin
   for k <– 0 to n-1 do
      for_all j <– k+1 to n-1 do
            A[k,j] <– A[k,j] / A[k,k];
      BARRIER(bar1, nprocs);
      for_all i <–k+1 to n-1
         for_all j <– k+1 to n-1 do
            A[i,j] <– A[i,j] - A[i,k]*A[k,j];
   endfor
```

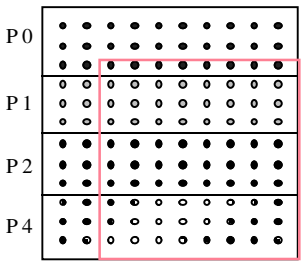
A is entire n-by-n shared matrix to be factored



© Culler 1995

Decomposition

- Work decomposition
  - **self scheduling**
    - » **each process increments a shared iteration variable**
  - **static scheduling**
    - » **formula to assign iterations**
- Block Row Decomposition
  - **task ~ ROW computation**
  - **pivot row scaling**
  - **update of lower-right submatrix**
- Assignment
  - **chunks of rows** (R = n/p)
  - **process** p **gets** p*R, ..., p*R+R-1



© Culler 1995

Self-Scheduling by rows

```
shared int i;

procedure LU( A[n,n] )
begin
   for k <– 0 to n-1 do
      if (pid == 0)
         for j <– k+1 to n-1 do
            A[k,j] <– A[k,j] / A[k,k];
         i <- k+1;
      endif
      BARRIER(bar1, nprocs);
      while ((ii <- fetch&inc(i)) < n-1)
         for j <– k+1 to n-1 do
            A[i,j] <– A[i,j] - A[i,k]*A[k,j];
   endfor
```


© Culler 1995

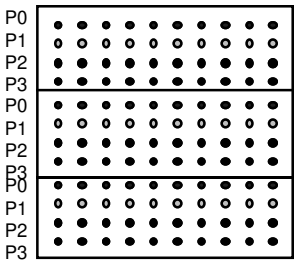
Block Row Assignment (ala ANL)

```
procedure LU( A[n, n] )          /* Global Shared Data Structure*/
begin
   int i,j,k,pid;                /* pid = process id in [0,nprocs-1], assigned when created*/
   int rows, mymin, mymax;       /* indices of first and last rows assigned to the  process */
   rows <– n / procs;                    /* assume it divides for simplicity */
   mymin <– pid * rows;   mymax <– mymin + rows - 1;
   for k <– 0 to n-1 do
      if (mymin ≤ k ≤ mymax)             /* if pivot row is among those assigned to me */
         for j <– k+1 to n-1 do          /* I scale row k */
            A[k,j] <– A[k,j] / A[k,k];
      endif
      BARRIER(bar1, nprocs);             /* every body else waits*/
      for i <– max(k+1,mymin) to mymax do  /* THE PARALLEL LOOP: a block of rows for me*/
         for j <– k+1 to n-1 do          /*The TASK: update a row */
            A[i,j] <– A[i,j] - A[i,k]*A[k,j];
   endfor
```


© Culler 1995

Alternative Decomposition

- Work decomposition
  - **task ~ ROW computation**
  - **pivot row scaling**
  - **update of lower-right submatrix**
- Assignment
  - **cyclic by rows** (R = n/p)
  - **process** p **gets** i **if** i mod R = p



© Culler 1995

Cyclic Row Assignment

```
procedure LU( A[n, n] )          /* Global Shared Data Structure*/
begin
   int i,j,k,pid;                /* pid = process id in [0,nprocs-1], assigned when created*/
   int rows, myfirst;            /* indices of first and last rows assigned to the  process */
   rows <– n / nprocs;                    /* assume it divides for simplicity */
   for k <– 0 to n -1 do
      if (k mod nprocs == pid) then       /* if pivot row is among those assigned to me */
         for j <– k+1 to n-1 do           /* I scale row k */
            A[k,j] <– A[k,j] / A[k,k];
      endif
      myfirst = ???
      BARRIER(bar1, nprocs);              /* every body else waits*/
      for i <– myfirst to n-1 by nprocs do /* THE PARALLEL LOOP: rows for me*/
         for j <– k+1 to n-1 do           /*The TASK: update a row */
            A[i,j] <– A[i,j] - A[i,k]*A[k,j];
   endfor
```


© Culler 1995

SPMD Msg Passing Example

```
int n, nprocs;                  /* matrix dimension and number of processors to be used
double **A';                    /* Local piece of Global Data structure

main()
begin
   read(n);       read(nprocs);   /* read input matrix size and number of processes
   nrows = n / nprocs;
   A' <– MALLOC (a two-dimensional array of size nrows x n doubles);
                                  /* create processes
   CREATE (nprocs-1 processes that start executing LU(A));
   LU(A);                         /* main process becomes a worker too
   WAIT_FOR_END;                  /* wait for all child processes created to terminate
end main

procedure LU( A'[nrows, n] )
   . . .
```

Decomposition of data is fundamental to algorithm
- **– dictates assignment of tasks**
- **– can only access local data**

© Culler 1995

Msg Passing Example

```
procedure LU( A'[ nrows, n ] )          /* Local rows of A
begin
double rowk[ n ] ;                       /* Local copy of row A[k,*]
   mymin <– pid * nrows;  mymax <– mymin + rows - 1;
   for k <– 0 to n-1 do                 /* outer loop over all diagonal elements */
      k' <– k mod rows;                 /* Local index of row k
      if (mymin ≤ k ≤ mymax) then       /* if pivot row is assigned to me */
         for j <– k+1 to n-1 do         /* scale the pivot row */
            A'[k',j] <– A'[k',j ] / A'[k',k];
         for p <– pid to nproc-1 do     /* Send it to all still active
            SEND(&myA[k',0], n*sizeof(double), p);
      endif
      if (k ≤ mymax) then               /* all active receive copy of row k from sender */
            RECEIVE(rowk, n*sizeof(double), *);
      for i <– max(k+1,mymin) to mymax do    /* for my active rows */
         i'  = i  mod rows
         for j <– k+1 to n-1 do              /* for all elements in that row */
            A'[i',  j] <– A'[i',j] - A'[i',k] * rowk[j];
```


© Culler 1995

Msg Passing with global operations

```
procedure LU( A'[ nrows, n ] )          /* Local rows of A
begin
double rowk[ n ] ;                      /* Local copy of row A[k,*]
    mymin <– pid * nrows;  mymax <– mymin + rows - 1;
    for k <– 0 to n-1 do                /* outer loop over all diagonal elements */
        k' <– k mod rows;               /* Local index of row k
        if (mymin ≤ k ≤ mymax) then     /* if pivot row is assigned to me */
            for j <– k+1 to n-1 do      /* scale the pivot row */
                A'[k',j] <– A'[k',j ] / A'[k',k];
        broadcast(PID, &myA[k',0], n*sizeof(double), rowK);
        for i <– max(k+1,mymin) to mymax do      /* for my active rows */
            i'  = i  mod rows
            for j <– k+1 to n-1 do               /* for all elements in that row */
                A'[i', j] <– A'[i',j] - A'[i',k] * rowk[j];
```


© Culler 1995

Shared Address with pt-to-pt synchronization

- Barrier built from simple operations

```
Barrier( *x, n )
    increment( x )          /* enter
    while ( *x < n) {};     /* wait (spin)
    x = o                   /* leave
```

- Atomic increment

```
    load r <- x
    add  r <- r + 1
    store x <- r

    lock( x )
    increment( x)
    unlock( x)
```

```
        r = -1              special lock val
L:      swap(r, X)
        if (r == -1) goto L
        add  r <- r + 1
        store x <- r
```

- What happens if one re-enters barrier before they all leave?

© Culler 1995

Shared Address with Flags

```
procedure LU( A[n, n], flags[n] )          /* Flags initialized to zero */
begin
   int i,j,k,pid;
   int rows, mymin, mymax;
   rows <– n / procs;
   mymin <– pid * rows;   mymax <– mymin + rows - 1;
   for k <– 0 to n-1 do
      if (mymin ≤ k ≤ mymax)
         for j <– k+1 to n-1 do
            A[k,j] <– A[k,j] / A[k,k];
            flag[k] <- 1                   /* Pivot row scaled
      endif
      while (flag[ k ] == 0) {};           /* Wait till pivot row is ready
      for i <– max(k+1,mymin) to mymax do
         for j <– k+1 to n-1 do
            A[i,j] <– A[i,j] - A[i,k]*A[k,j];
   endfor
```


© Culler 1995

Distributed Shared Address Space

- HPF: Fortran90 (data parallel) + layout
- Fortran-D: F77 + layout + parallelization
- Several parallel C

```
procedure LU( A[n,n] )
distribution A [ CYCLIC, *]
begin
   for k <– 0 to n-1 do
      for_all j <– k+1 to n-1 do
            A[k,j] <– A[k,j] / A[k,k];
      BARRIER(bar1, nprocs);
      for_all i <–k+1 to n-1
         for_all j <– k+1 to n-1 do
            A[i,j] <– A[i,j] - A[i,k]*A[k,j];
   endfor
```

in Split-C
A[n]::[n]

Declaration tells compiler how to allocate storage for global object

Compiler can figure out how to distribute the loops

© Culler 1995

Elements of Parallel Programs

- Process creation
- Allocation of shared data
  - **external static data**
  - **global malloc**
- Assignment of work
  - **loop distribution**
  - **task queues**
- Synchronization
  - **all-to-all events**
    - » **barrier, wait_for_end**
  - **point-to-point events**
    - » **flags**
    - » **messages**
  - **mutual exclusion**
    - » **lock, unlock**

© Culler 1995

Comparison

- Shared Memory
  - **Explicit global data structure**
  - **Decomposition of work is independent of data layout**
  - **Communication is implicit**
  - **Explicit synchronization**
    - » **need to avoid race condition and over-writing**
- Message Passing
  - **Implicit global data structure**
    - » **consider row cyclic version !!!**
  - **Decomposition of data determines assignment of work**
  - **Communication is explicit**
  - **Synchronization is implicit**
    - » **data buffered till received**

© Culler 1995

Decomposition and Assignment

- Determine how work is divided up among collaborating processes
  - **structural approaches**
    - » **parallel loops or loop nests**
  - **application domain decomposition**
  - **data decomposition**
  - **task queues**
- Usually adopt the view: process <-> processor
  - **multiprogramming?**
  - **dynamic resources?**

© Culler 1995

Orchestration

- data access, communication, and synchronization among processes performing assigned tasks
- shared address space
  - **definition of shared variables and data structures**
    - » **global structures are explicit**
    - » **communication is implicit in access patterns**
  - **mutual exclusion and events through atomic operations on shared variables**
    - » **synchronization is distinct from communication**
- message passing
  - **program data is distributed among local address spaces**
    - » **global structures are implicit in communication patterns**
    - » **communication is explicit**
  - **mutual exclusion by fiat**
  - **event synchronization is inherent in communication**
- barriers and point-to-point events to enforce dependences

© Culler 1995

What limits the performance of a parallel program?

- Available Parallelism
- Load Balance
  - **some processors work while others wait**
- Extra work
  - **management of parallelism**
  - **redundant computation**
- Communication

© Culler 1995

Exercise: For row-based LUD

- What is the available parallelism?
  - **as a function of problem size?**
- How bad is the load balance?
  - **blocks of rows?**
  - **cyclic rows?**
- Parallelism overhead?
- Communication?

© Culler 1995