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

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

May be represented by a good sequential program

Problem

Decomposition

Tasks

Processes

Orchestration

Assig

ment

May be represented by a good sequential program
Goal: sufficiency, not optimality

- Programming perspective
  - high performance
  - low resource usage
  - low development time

- Architecture perspective
  - high performance
  - low hardware complexity
  - low development time

Example: LU Decomposition

Simplification of the core of LINPACK benchmark.
Sequential Program (by rows)

```c
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 */
begin
    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*/
            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 */
    endfor
endfor
```

Identify Concurrency
Programming Approaches

### Programming Model

- **Message Passing**
- **Shared Address Space**
  - **Global**
  - **Synchronization**
  - **Uniform Distributed**
  - **pt-to-pt**

### Cost Model

- **Traditional MPP**
- **SMP**
- **Uni**
- **SMP**

### SPMD Shared Address

```c
int n, nprocs;
/* matrix dimension and number of processors to be used
double **A;
BARDEC (bar1);
/* Global Data structure
/* 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));
  /* 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
Shared Address, uniform, global

procedure LU( A[n,n] ) A is entire n-by-n shared matrix to be factored
begin
  for k <- 0 to n-1 do
    for_all j <- k+1 to n-1 do
      BARRIER(bar1, nprocs);
    endfor
    for_all i <- k+1 to n-1
      for_all j <- k+1 to n-1 do
      endfor
  endfor
endLU

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
Self-Scheduling by rows

```plaintext
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
                i <- k+1;
            endif
        endif
        BARRIER(bar1, nprocs);
        while ((ii <- fetch&inc(i)) < n-1)
            for j <- k+1 to n-1 do
            endfor
        endwhile
    endfor

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 */
            endfor
        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 */
            endfor
        endfor
    endfor
```
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

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 */
            endfor
    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 */
        endfor
    endfor
end
```
SPMD Msg Passing Example

```c
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);
    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

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), *);
            endif
        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];
```
Msg Passing with global operations

```plaintext
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];
end
```

Shared Address with pt-to-pt synchronization

- Barrier built from simple operations

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

- Atomic increment

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

L: swap(r, X)

```
if (r == -1) goto L
increment( x)
unlock( x)
```

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

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
      BARRIER(bar1, nprocs);
    endfor
  for all i <- k+1 to n-1 do
    for all j <- k+1 to n-1 do
  endfor
in Split-C
A[n]:[n]
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

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
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?

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
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

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?