Module 7: "Parallel Programming"

Lecture 13: "Parallelizing a Sequential Program"

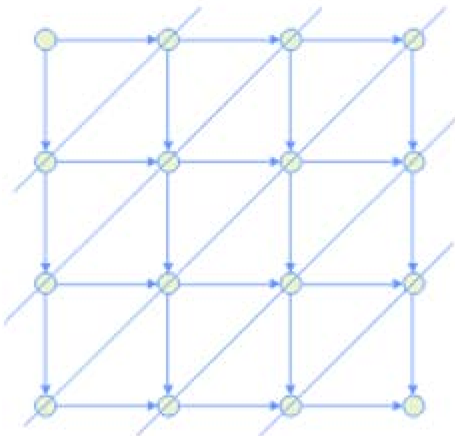
Parallel Programming

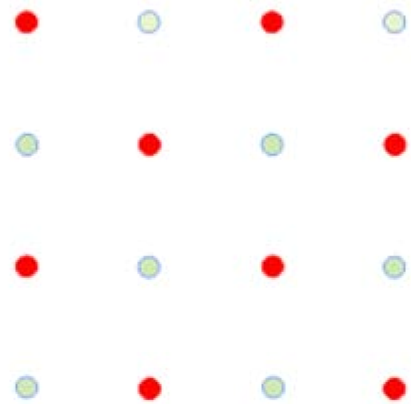
- Decomposition of Iterative Equation Solver
- Assignment
- Shared memory version
- Mutual exclusion
- LOCK optimization
- More synchronization
- Message passing
- Major changes
- Message passing
- Message Passing Grid Solver
- MPI-like environment

[From Chapter 2 of Culler, Singh, Gupta]

Decomposition of Iterative Equation Solver

- Look for concurrency in loop iterations
  - In this case iterations are really dependent
  - Iteration (i, j) depends on iterations (i, j-1) and (i-1, j)





  - Each anti-diagonal can be computed in parallel
  - Must synchronize after each anti-diagonal (or pt-to-pt)
  - Alternative: red-black ordering (different update pattern)
- Can update all red points first, synchronize globally with a barrier and then update all black points
  - May converge faster or slower compared to sequential program
  - Converged equilibrium may also be different if there are multiple solutions
  - Ocean simulation uses this decomposition
- We will ignore the loop-carried dependence and go ahead with a straight-forward loop decomposition
  - Allow updates to all points in parallel
  - This is yet another different update order and may affect convergence
  - Update to a point may or may not see the new updates to the nearest neighbors (this parallel algorithm is non-deterministic)

```
while (!done)
  diff = 0.0;
  for_all i = 0 to n-1
    for_all j = 0 to n-1
      temp = A[i, j];
      A[i, j] = 0.2(A[i, j]+A[i, j+1]+A[i, j-1]+A[i-1, j]+A[i+1, j]);
      diff += fabs (A[i, j] – temp);
    end for_all
  end for_all
  if (diff/(n*n) < TOL) then done = 1;
end while
```

- Offers concurrency across elements: degree of concurrency is n^2
- Make the j loop sequential to have row-wise decomposition: degree n concurrency

Module 7: "Parallel Programming"

Lecture 13: "Parallelizing a Sequential Program"

Assignment

- Possible static assignment: block row decomposition
  - Process 0 gets rows 0 to (n/p)-1, process 1 gets rows n/p to (2n/p)-1 etc.
- Another static assignment: cyclic row decomposition
  - Process 0 gets rows 0, p, 2p,…; process 1 gets rows 1, p+1, 2p+1,….
- Dynamic assignment
  - Grab next available row, work on that, grab a new row,…
- Static block row assignment minimizes nearest neighbor communication by assigning contiguous rows to the same process

Shared memory version

```
/* include files */
MAIN_ENV;
int P, n;
void Solve ();
struct gm_t {
  LOCKDEC (diff_lock);
  BARDEC (barrier);
  float **A, diff;
} *gm;
int main (char **argv, int argc)
{
  int i;
  MAIN_INITENV;
  gm = (struct gm_t*) G_MALLOC (sizeof (struct gm_t));
  LOCKINIT (gm->diff_lock);
BARINIT (gm->barrier);
  n = atoi (argv[1]);
  P = atoi (argv[2]);
  gm->A = (float**) G_MALLOC ((n+2)*sizeof (float*));
  for (i = 0; i < n+2; i++) {
    gm->A[i] = (float*) G_MALLOC ((n+2)*sizeof (float));
  }
  Initialize (gm->A);
  for (i = 1; i < P; i++) {  /* starts at 1 */
    CREATE (Solve);
  }
  Solve ();
  WAIT_FOR_END (P-1);
  MAIN_END;
}
```

```
void Solve (void)
{
  int i, j, pid, done = 0;
```

```
    float temp, local_diff;
    GET_PID (pid);
    while (!done) {
       local_diff = 0.0;
       if (!pid) gm->diff = 0.0;
       BARRIER (gm->barrier, P);/*why?*/
       for (i = pid*(n/P); i < (pid+1)*(n/P); i++) {
          for (j = 0; j < n; j++) {
             temp = gm->A[i] [j];
            gm->A[i] [j] = 0.2*(gm->A[i] [j] + gm->A[i] [j-1] + gm->A[i] [j+1] + gm->A[i+1] [j] + gm->A[i-1]
[j]);
local_diff += fabs (gm->A[i] [j] – temp);
          }   /* end for */
       }    /* end for */
       LOCK (gm->diff_lock);
       gm->diff += local_diff;
       UNLOCK (gm->diff_lock);
       BARRIER (gm->barrier, P);
       if (gm->diff/(n*n) < TOL) done = 1;
       BARRIER (gm->barrier, P); /* why? */
    }   /* end while */
}
```

Previous Next

Mutual exclusion

- Use LOCK/UNLOCK around critical sections
  - Updates to shared variable diff must be sequential
  - Heavily contended locks may degrade performance
  - Try to minimize the use of critical sections: they are sequential anyway and will limit speedup
  - This is the reason for using a local_diff instead of accessing gm->diff every time
  - Also, minimize the size of critical section because the longer you hold the lock, longer will be the waiting time for other processors at lock acquire

LOCK optimization

- Suppose each processor updates a shared variable holding a global cost value, only if its local cost is less than the global cost: found frequently in minimization problems

```
LOCK (gm->cost_lock);
if (my_cost < gm->cost) {
gm->cost = my_cost;
}
UNLOCK (gm->cost_lock);
/* May lead to heavy lock contention if everyone tries to update at the same time */

if (my_cost < gm->cost) {
LOCK (gm->cost_lock);
if (my_cost < gm->cost)
{ /* make sure*/
gm->cost = my_cost;
}
UNLOCK (gm->cost_lock);
} /* this works because gm->cost is monotonically decreasing */
```

More synchronization

- Global synchronization
  - Through barriers
  - Often used to separate computation phases
- Point-to-point synchronization
  - A process directly notifies another about a certain event on which the latter was waiting
  - Producer-consumer communication pattern
  - Semaphores are used for concurrent programming on uniprocessor through P and V functions
  - Normally implemented through flags on shared memory multiprocessors (busy wait or spin)

P_0: A = 1; flag = 1;

P_1: while (!flag); use (A);

Module 7: "Parallel Programming"

Lecture 13: "Parallelizing a Sequential Program"

Message passing

- What is different from shared memory?
  - No shared variable: expose communication through send/receive
  - No lock or barrier primitive
  - Must implement synchronization through send/receive
- Grid solver example
  - P_0 allocates and initializes matrix A in its local memory
  - Then it sends the block rows, n, P to each processor i.e. P_1 waits to receive rows n/P to 2n/P-1 etc. (this is one-time)
  - Within the while loop the first thing that every processor does is to send its first and last rows to the upper and the lower processors (corner cases need to be handled)
  - Then each processor waits to receive the neighboring two rows from the upper and the lower processors
- At the end of the loop each processor sends its *local_diff* to P_0 and P_0 sends back the done flag

Major changes

/* include files */
MAIN_ENV;
~~int P, n;~~
void Solve ();
~~struct gm_t {~~
  ~~LOCKDEC (diff_lock);~~
  ~~BARDEC (barrier);~~
  ~~float **A, diff;~~
~~} *gm;~~

int main (char **argv, int argc)
{
  int i; **int P, n; float **A;**
  MAIN_INITENV;
  ~~gm = (struct gm_t*) G_MALLOC (sizeof (struct gm_t));~~
  ~~LOCKINIT (gm->diff_lock);~~
  ~~BARINIT (gm->barrier);~~
  n = atoi (argv[1]);
  P = atoi (argv[2]);

Local Alloc.

  gm->A = (float**) G_MALLOC ((n+2)*sizeof (float*));
  for (i = 0; i < n+2; i++) {
    gm->A[i] = (float*) G_MALLOC ((n+2)*sizeof (float));
  }
  Initialize (gm->A);
  for (i = 1; i < P; i++) { /* starts at 1 */
    CREATE (Solve);
  }
  Solve ();
  WAIT_FOR_END (P-1);
  MAIN_END;
}

```
void Solve (void)
{
  int i, j, pid, done = 0;
  float temp, local_diff;
  GET_PID (pid);
  while (!done) {
    local_diff = 0.0;
    if (!pid) gm->diff = 0.0;
    BARRIER (gm->barrier, P);/*why?*/
    for (i = pid*(n/P); i < (pid+1)*(n/P);
    I++) {
      for (j = 0; j < n; j++) {
        temp = gm->A[i] [j];
        gm->A[i] [j] = 0.2*(gm->A[i] [j] +
        gm->A[i] [j-1] + gm->A[i] [j+1] + gm-
        >A[i+1] [j] + gm->A[i-1] [j];
          local_diff += fabs (gm->A[i] [j] –
temp);
      } /* end for */
    } /* end for */
    LOCK (gm->diff_lock);
    gm->diff += local_diff;
    UNLOCK (gm->diff_lock);
    BARRIER (gm->barrier, P);
    if (gm->diff/(n*n) < TOL) done = 1;
    BARRIER (gm->barrier, P); /* why? */
  } /* end while */
}
```

if (pid) Recv rows, n, P

Send up/down
Recv up/down

Send local diff
to P_0
Recv diff

Previous Next

Message passing

- This algorithm is deterministic
- May converge to a different solution compared to the shared memory version if there are multiple solutions: why?
    - There is a fixed specific point in the program (at the beginning of each iteration) when the neighboring rows are communicated
    - This is not true for shared memory

Message Passing Grid Solver

MPI-like environment

- MPI stands for Message Passing Interface
    - A C library that provides a set of message passing primitives (e.g., send, receive, broadcast etc.) to the user
- PVM (Parallel Virtual Machine) is another well-known platform for message passing programming
- Background in MPI is not necessary for understanding this lecture
- Only need to know
    - When you start an MPI program every thread runs the same main function
    - We will assume that we pin one thread to one processor just as we did in shared memory
- Instead of using the exact MPI syntax we will use some macros that call the MPI functions

```
MAIN_ENV;
/* define message tags */
 #define ROW 99
#define DIFF 98
#define DONE 97
int main(int argc, char **argv)
{
   int pid, P, done, i, j, N;
   float tempdiff, local_diff, temp, **A;
   MAIN_INITENV;
   GET_PID(pid);
   GET_NUMPROCS(P);
   N = atoi(argv[1]);
   tempdiff = 0.0;
   done = 0;
   A = (double **) malloc ((N/P+2) * sizeof(float *));
   for (i=0; i < N/P+2; i++) {
       A[i] = (float *) malloc (sizeof(float) * (N+2));
   }
   initialize(A);
while (!done) {
    local_diff = 0.0;
   /* MPI_CHAR means raw byte format */
```

```
if (pid) {  /* send my first row up */
   SEND(&A[1][1], N*sizeof(float), MPI_CHAR, pid-1, ROW);
}
if (pid != P-1) {  /* recv last row */
   RECV(&A[N/P+1][1], N*sizeof(float), MPI_CHAR, pid+1, ROW);
}
if (pid != P-1) {  /* send last row down */
   SEND(&A[N/P][1], N*sizeof(float), MPI_CHAR, pid+1, ROW);
}
if (pid) {  /* recv first row from above */
   RECV(&A[0][1], N*sizeof(float), MPI_CHAR, pid-1, ROW);
}
for (i=1; i <= N/P; i++) for (j=1; j <= N; j++) {
      temp = A[i][j];
      A[i][j] = 0.2 * (A[i][j] + A[i][j-1] +           A[i-1][j] + A[i][j+1] + A[i+1][j]);
      local_diff += fabs(A[i][j] - temp);
     }
if (pid) {  /* tell P0 my diff */
      SEND(&local_diff, sizeof(float),    MPI_CHAR, 0, DIFF);
       RECV(&done, sizeof(int), MPI_CHAR, 0, DONE);
   }
   else {  /* recv from all and add up */
      for (i=1; i < P; i++) {
         RECV(&tempdiff, sizeof(float), MPI_CHAR, MPI_ANY_SOURCE, DIFF);
         local_diff += tempdiff;
      }
      if (local_diff/(N*N) < TOL) done=1;
      for (i=1; i < P; i++) {
         /* tell all if done */
         SEND(&done, sizeof(int), MPI_CHAR, i, DONE);
      }
   }
}  /* end while */
MAIN_END;
}  /* end main */
```

- Note the matching tags in SEND and RECV
- Macros used in this program
  - GET_PID
  - GET_NUMPROCS
  - SEND
  - RECV
- These will get expanded into specific MPI library calls
- Syntax of SEND/RECV
  - Starting address, how many elements, type of each element (we have used byte only), source/dest, message tag

Previous Next