Lecture 7: Examples of Shared Memory and Message Passing Programming

The Lecture Contains:

Previous Next

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

```
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 */
```

Previous Next

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)

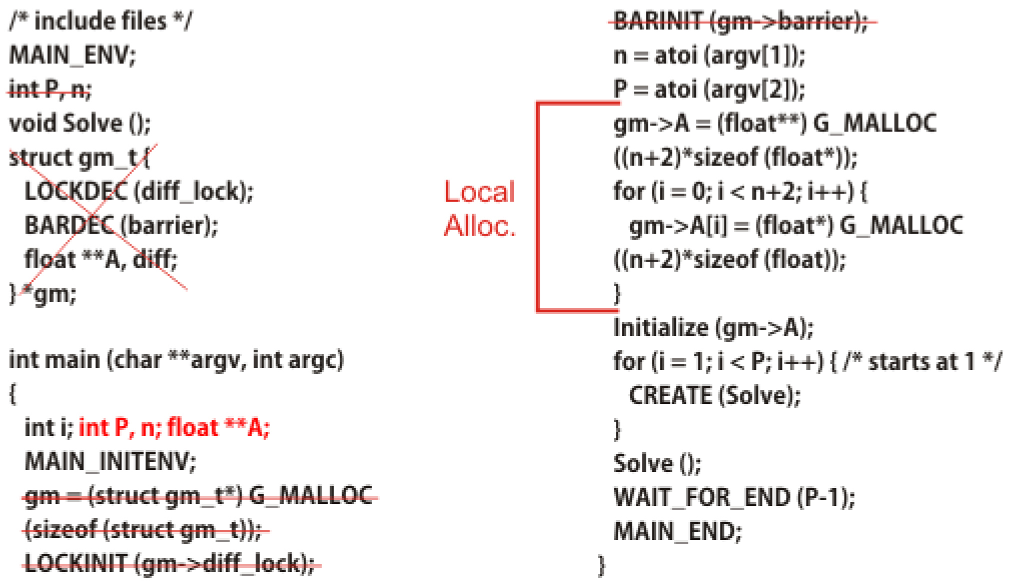
P 0 : A = 1; flag = 1;
P 1 : while (!flag); use (A);

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 accumulated diff so that each processor can locally compute the done flag

Previous Next

Major Changes

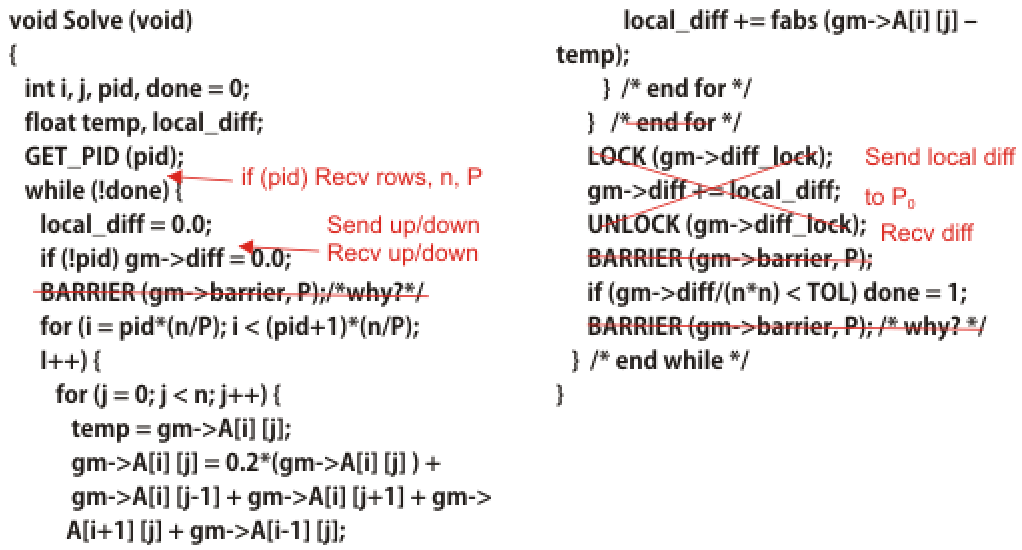


```
/* include files */
MAIN_ENV;
~~int P, n;~~
void Solve ();
~~struct gm_t {~~
  ~~LOCKDEC (diff_lock);~~
  ~~BARDEC (barrier);~~
  ~~float **A, diff;~~
~~} *gm;~~

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]);
  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;
}
```

Previous Next

Major Changes



```
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 */
}
```

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

Previous Next

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);
}
```

Previous Next