Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2023-24
«MPI Programming Model – III: Master-Worker Execution»

Παναγιώτης Χατζηδούκας, Ευστράτιος Γαλλόπουλος

Outline

- Hands-on: Task queue in OpenMP
- Hands-on: Task queue in MPI (master-worker)

I. Sequential Code

```
void task(double *x, double *y) {
      *y = x[0]+x[1];
}

int main(int argc, char *argv[]) {
      double result[100];

      for (int i=0; i<100; i++) {
          double d[2];
          d[0] = drand48();
          d[1] = drand48();
          task(d, &result[i]);
      }

       /* print results */
       return 0;
}
```

OpenMP Code

```
void task(double *x, double *y) { *y = x[0] + x[1]; }

int main(int argc, char *argv[]) {
      double result[100];

      #pragma omp parallel
      #pragma omp single nowait
      {
          for (int i=0; i<100; i++) {
              double d[2];
              d[0] = drand48();
              d[1] = drand48();
              #pragma omp task firstprivate(d, i) shared(result)
              {
                task(d, &result[i]);
              }
          }
          #pragma omp taskwait
          /* print results */
      }
       return 0;
}
```

OpenMP + Task Queue (1/4)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

void master();
void worker();

int main(int argc, char *argv[])
{
    int  myrank;

    #pragma omp parallel
    {
        myrank = omp_get_thread_num();

        if (myrank == 0) {
            master();
        } else {
            worker();
        }
    }
    return 0;

}
```

OpenMP + Task Queue (2/4)

```
typedef struct work_s
{
    int pos;
    double x[2];
    double y;
} work_t;

work_t *workarray;
int work_id, nworks;

void init_work(int n)
{
    int i;

    #pragma omp critical
    {
        workarray = malloc(n*sizeof(work_t));
        work_id = 0;
        nworks = n;
        srand48(1);
        for (i = 0; i < n; i++) {
            workarray[i].pos = i;
            workarray[i].x[0] = drand48();
            workarray[i].x[1] = drand48();
        }
    }
}
```

OpenMP + Task Queue (3/4)

```
work_t *get_next_work_request()
{
    int local_work_id;

    #pragma omp critical
    {
        local_work_id = work_id++;
    }

    if (local_work_id >= nworks) return NULL;

    work_t *w = &workarray[local_work_id];

    return w;
}

void print_work()
{
    int i;
    int n = nworks;

    for (i = 0; i < n; i++) {
        printf("work[%d]: %f %f -> %f\n", i,
                workarray[i].x[0], workarray[i].x[1], workarray[i].y);
    }

}
```

OpenMP + Task Queue (4/4)

```
void master()
{
    init_work(10);

    worker();

    /* Print the results */
    print_work();
}

void worker()
{
    work_t                  *work;

    #pragma omp barrier

    work = get_next_work_request();
    while (work != NULL)
    {
        work->y   = work->x[0] + work->x[1];
        sleep(1);
        printf("result of work %d on %d : %f\n", work->pos, omp_get_thread_num(), work->y);
        work = get_next_work_request();
    }

    #pragma omp barrier
}
```

II. MPI Master-Worker (1/7)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define WORKTAG       1
#define DIETAG        2

void master();
void worker();

int main(int argc, char *argv[])
{
    int  myrank;

    MPI_Init(&argc, &argv);                  /* initialize MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);      /* process rank, 0 to N-1 */
    if (myrank == 0) {
        master();
    } else {
        worker();
    }
    MPI_Finalize();        /* cleanup MPI */

    return 0;
}
```

MPI Master-Worker (2/7)

```
/* A very naive work queue */

typedef struct work_s
{
     int pos;
     double x[2];
     double y;
} work_t;

work_t *workarray;
int work_id, nworks;

void init_work(int n)
{
     int i;

     workarray = malloc(n*sizeof(work_t));
     work_id = 0;
     nworks = n;
     for (i = 0; i < n; i++) {
          workarray[i].pos = i;
          workarray[i].x[0] = drand48();
          workarray[i].x[1] = drand48();
     }
}
```

MPI Master-Worker (3/7)

```
work_t *get_next_work_request()
{
    if (work_id >= nworks) return NULL;

    work_t *w = &workarray[work_id];
    work_id++;

    return w;
}

void print_work()
{
    int i;
    int n = nworks;

    for (i = 0; i < n; i++) {
        printf("work[%d]: %f %f -> %f\n", i, workarray[i].x[0],
                 workarray[i].x[1], workarray[i].y);
    }
}
```

MPI Master-Worker (4/7)

```
typedef struct result_s
{
    int pos;
    double y;
} result_t;


void worker()
{
    int rank;
    result_t            result;
    work_t              work;
    MPI_Status          status;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (;;) {
        MPI_Recv(&work, sizeof(work_t), MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

        /* Check the tag of the received message */
        if (status.MPI_TAG == DIETAG) {
            return;
        }

        result.pos = work.pos;
        result.y  = work.x[0] + work.x[1];
        sleep(1);

        printf("result of work %d on %d : %f\n", result.pos, rank, result.y);
        MPI_Send(&result, sizeof(result_t), MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
}
```

MPI Master-Worker (5/7)

```
void master()
{
    int  ntasks, rank;
    double       result;
    MPI_Status      status;
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);           /* #processes in application */

    init_work(10);

    /*
     * Seed the workers.
     */

    work_t *work;
    for (rank = 1; rank < ntasks; ++rank) {
         work = get_next_work_request();

         printf("sending work %d\n", work->pos);

         MPI_Send(work,         /* message buffer */
             sizeof(work_t),        /* one data item */
             MPI_CHAR,          /* data item is a struct */
             rank,              /* destination process rank */
             WORKTAG,      /* user chosen message tag */
             MPI_COMM_WORLD);   /* always use this */
    }

    /* … */
```

MPI Master-Worker (6/7)

```
/*
 * Receive a result from any worker and dispatch a new work
 * until work requests have been exhausted.
 */

result_t res;
work = get_next_work_request();
while (work != NULL) {
    MPI_Recv(&res,                  /* message buffer */
         sizeof(result_t),          /* one data item .. */
         MPI_CHAR,                  /* of a struct  */
         MPI_ANY_SOURCE,                /* receive from any sender */
         MPI_ANY_TAG,               /* any type of message */
         MPI_COMM_WORLD,                /* always use this */
         &status);                  /* received message info */

    workarray[res.pos].y = res.y;

    printf("sending work %d\n", work->pos);
    MPI_Send(work, sizeof(work_t), MPI_CHAR, status.MPI_SOURCE, WORKTAG, MPI_COMM_WORLD);
    work = get_next_work_request();
}

/* … */
```

MPI Master-Worker (7/7)

```
    /*
     * Receive results for pending work requests.
     */
    for (rank = 1; rank < ntasks; ++rank) {
        MPI_Recv(&res, sizeof(result_t), MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
      workarray[res.pos].y = res.y;
    }

    /*
     * Tell all the workers to exit.
     */
    for (rank = 1; rank < ntasks; ++rank) {
        MPI_Send(0, 0, MPI_CHAR, rank, DIETAG, MPI_COMM_WORLD);
    }


    /* Print the results */
    print_work();
}
```