一、填空题

1. 常用的 Linpack 基准测试程序有 HPL 和 HPCC 。
2. MPI 的英文全称为 Message-Passing Interface ；
   集合通信的英文为 Collective communication 。
3. MPI 的通信分为 点对点通信 和 集合通信 。
4. 请写出在 3 个主机上启动 MPD 进程，形成 MPI 运行时的环境的命令：
   mpdboot -n 3 -f mpd.hosts 。（注目标主机列表由 $HOME/mpd.hosts 文件制定 ）
5. 只有接收进程可以使用通配符参数，常见的通配符参数包括 MPI_ANY_SOURCE 和 MPI_ANY_TAG。
6. OpenMP 适用于 共享内存 系统进行并行程序设计，其中#pragma omp critical 的功能是 保护临界区，使同一时间内只能有一个线程执行临界区代码 。
7. OpenMP 中，最主要是的编译指导语句，一条编译指导语句由 指令 和 子句 组成。
8. 在 Pthreads 中，引起竞争的代码称为临界区；常用来保护临界区的方式包括： 忙等待 、 互斥量 和信号量。
9. MPI_Scatter(a,local_n,MPI_DOUBLE,local_a,local_n,MPI_DOUBLE,0,comm)这条语句执行的功能 将进程 0 中数组 a 的各元素按照 local_n 的块大小以块划分的方式发送给通信子 comm 内的所有进程。
10. #pragma omp parallel num_threads(thread_count) reduction(+:global_result) 这条编译性指导语句的作用是 启动 thread_count 个线程并行执行代码块，并将代码块中的 global_result 作为规约变量，其中规约操作为加法（注意规约变量为共享作用域，但在各进行内会创建其私有副本） 。

二、简答题

1. 请简述高性能计算。

高性能计算(High performance computing)是指通常使用很多处理器（作为单个机

器的一部分）或者某一集群中组织的几台计算机（作为单个计算资源操作）的计算系统和环境。

2. 在linux系统中，对文件名为hello.c的MPI源程序、pthreads源程序和OpenMP源程序分别写出采用3个进程或线程的MPI、Pthreads和OpenMP的编译和运行的命令。

答：

MPI的编译和运行的命令分别为：

mpicc -g -Wall -o mpi_hello hello.c

mpiexec -n 3 mpi_hello

Pthreads的编译和运行的命令分别为：

gcc −g −Wall −o pth_hello hello . c −lpthread

./ pth_hello 3

OpenMP的编译和运行的命令分别为：

gcc −g −Wall −fopenmp −o omp_hello hello.c

./ omp_example 3

3. 考虑下面的循环

```
a[0]=0;
for(i=1;i<n;i++)
    a[i]=a[i-1]+2i;
```

在这个循环中显然有循环依赖，因为在计算a[i]前必须先算a[i-1]的值。请找一种方法消除循环依赖，并且并行化这个循环。

答：

观察如下的规律：

a[0] = 0

a[1] = a[0] + 2 = 0 + 2

a[2] = a[1] + 4 = 0 + 2 + 4

a[3] = a[2] + 6 = 0 + 2 + 4 + 6

a[4] = a[3] + 8 = 0 + 2 + 4 + 6 + 8

由此可得，$a[i]=2\sum_{j=0}^{i} j=i(i+1)$；

所以我们能够重新写代码为

for(i=0;i<n;i++)

```
    a[i]=i*(i+1);
```

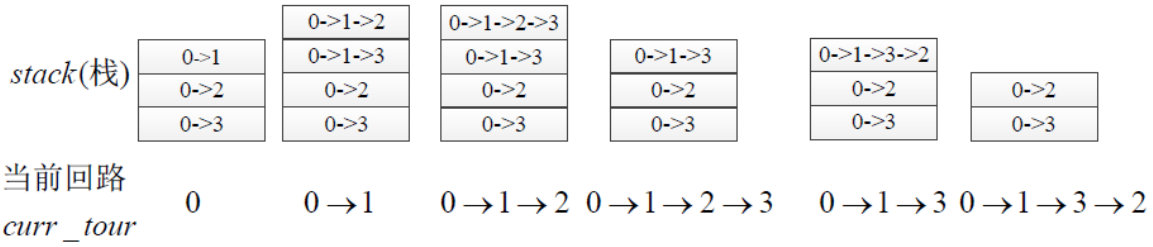
在此循环中，任何迭代的结果都不会再次使用。因此，代码可以与指令的并行化：

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared(a, n)
for (i = 0; i < n; i++)
     a[i] = i*(i+1);
```

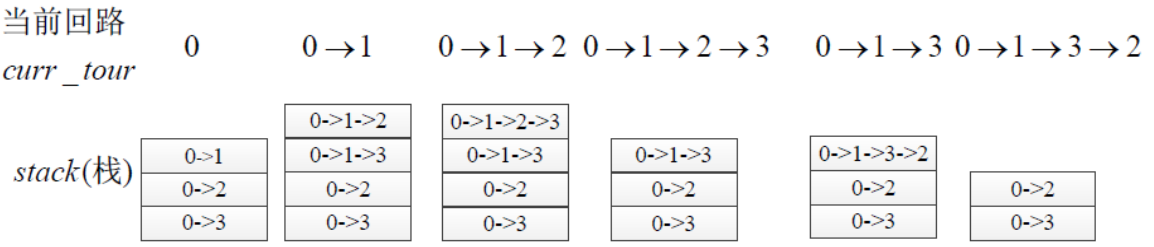
4. 下述代码是非递归深度优先搜索的代码片段，并假设Feasible(curr_tour, nbr)函数始终可行（为真）。现考虑4个城市的TSP问题，请画出栈和当前回路的运行状态，直到当前回路curr_tour为$0\rightarrow1\rightarrow3\rightarrow2$；请找出下述并行代码的临界区，并采用OpenMP的方式解决临界区问题。

```
Push_copy(stack,tour);//将回路压入栈中
while(!Empty(stack)){
    curr_tour=Pop(stack);//将回路弹出栈中
    if(City_count(curr_tour)==n){
        if(Best_tour(curr_tour))
            Update_best_tour(curr_tour);
    }else{
         for (nbr = n-1; nbr >= 1; nbr--)
            if (Feasible(curr_tour, nbr)) {
                Add_city(curr_tour, nbr);
                Push_copy(stack, curr_tour, avail);
                Remove_last_city(curr_tour);
            }
    }
    Free_tour(curr_tour);
}
```

答：



或者



并行代码的临界区为：

Update_best_tour(curr_tour);

采用 OpenMP 的方式保护临界区：

#pragma omp critical

Update_best_tour(curr_tour);

5. 判断下列程序是否能够安全通信，且说明原因。若如果不安全，请说明原因并给出相应的解决方案。

```
comm=MPI_COMM_WORLD;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0)
{     MPI_Recv(x2 , 3 , MPI_INT,1,tag,comm,&status);//A 语句
      MPI_Send(x1 , 3 , MPI_INT,1,tag,comm);//B 语句    }
if(rank==1)
{     MPI_Recv(x1, 3 ,MPI_INT,0,tag,comm,&status);//C 语句
      MPI_Send(x2, 3 ,MPI_INT,0,tag,comm);//D 语句    }
……
```

答案：不能够安全通信。

发送消息可能是阻塞或缓冲，而接收消息始终是阻塞。由于进程0和进程1都是先接收消息，后发送消息，所以进程 0 和 1 都会阻塞等待，此时没有任何进行发送消息。因此，上述代码会发送死锁，不能够安全通信。

解决方案为：将 A 语句与 B 语句进行调换顺序；或者 C 语句与 D 语句进行调换顺序。

二、程序阅读题

1. 补全如下的代码片段以实现 MPI 程序进行计时并报告运行时间。

```
double local_start, local_finish,local_elapsed,elapsed;// elapsed 存储并行程序的运行时间
…
   MPI_Barrier(comm) ;
   local_start=MPI_Wtime();
/* Code to be timed */
local_finish =MPI_Wtime();
local_elapsed=local_finish-local_start;
MPI_Reduce(&local_elapsed,&elapsed,1,MPI_DOUBLE,  MPI_MAX,0,comm);
//注通信子为 comm
if (my_rank ==0)
     printf(“Elapsed time = %e seconds\n”,elapsed);
```

2.写出下述程序的运行结果。

```
int k=100;
#pragma omp parallel for private(k)
for(i=0;i<2;i++)
{
      k+=i;
      printf(“k=%d\n”,k);
}
printf(“last k=%d\n”,k);
```

运行结果为：

```
k=0
k=1
last k=100
```

。

j=0, ThreadId=0

3. 写出以下程序段的运行结果：j=1, ThreadId=1，若将#pragma omp parallel num_threads(2){}

j=0, ThreadId=0

删除，程序段的运行结果：j=1, ThreadId=0。

```
int j=0;
#pragma omp parallel num_threads(2)
{
#pragma omp for
for(j=0;j<2;j++)
    printf("j=%d, ThreadId=%d\n",j,omp_get_thread_num());}
```

四、程序题

1. 请将以下程序改写为 10 个线程执行的 OpenMP 并行程序，以实现使用如下的公式对π进行估计的功能

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    long long n, i;
    double factor;
    double sum = 0.0;
    n=1000;
    for (i = 0; i < n; i++) {
        sum += factor/(2*i+1);
        factor=-factor;
    }
    sum = 4.0*sum;
    printf("Our estimate of pi = %.14f\n", sum);
    return 0;
```

```
}   /* main */
```

答：

```
int main(int argc, char* argv[]) {
    long long n, i;
    int thread_count=10;
    double factor;
    double sum = 0.0;
    n=1000;
#   pragma omp parallel for num_threads(thread_count) reduction(+: sum) private(factor)
    for (i = 0; i < n; i++) {
        factor = (i % 2 == 0) ? 1.0 : -1.0;
        sum += factor/(2*i+1);
    }
    sum = 4.0*sum;
    printf("With n = %lld terms and %d threads,\n", n, thread_count);
    printf("      Our estimate of pi = %.14f\n", sum);
    return 0;
}   /* main */
```

2. 采用 MPI 实现如下的梯形积分，即估计函数 $y = f(x)$ 在区间 $[a,b]$ 上与 x 所围阴影部分的面积，如下图所示。

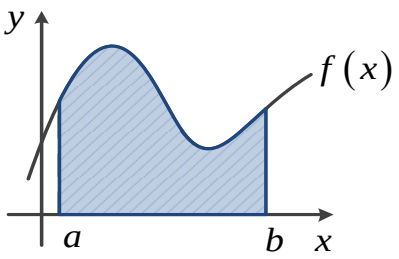


图 1 梯形积分

完成如下要求：

1) 主函数：采用**点对点通信**，即 0 号进程接收消息并打印，其它进程发送消息；

2) **接收用户输入函数 Get_input**：采用广播 MPI_Bcast 函数实现。

注意：采用如下已编写的梯形积分函数 Trap(double left_endpt, double right_endpt, int trap_count, double base_len)。

```
double Trap(
        double left_endpt    /* in */,
        double right_endpt /* in */,
        int       trap_count    /* in */,
        double base_len       /* in */) {
```

```
    double estimate, x;
    int i;
    estimate = (f(left_endpt) + f(right_endpt))/2.0;
    for (i = 1; i <= trap_count-1; i++) {
        x = left_endpt + i*base_len;
        estimate += f(x);
    }
    estimate = estimate*base_len;
    return estimate;
} /*  Trap  */
```

答 1）

程序 1 main 函数

```
int main(void) {
    int my_rank, comm_sz, n, local_n;
    double a, b, h, local_a, local_b;
    double local_int, total_int;
    int source;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    Get_input(my_rank, comm_sz, &a, &b, &n);
    h = (b-a)/n;                /* h is the same for all processes */
    local_n = n/comm_sz;    /* So is the number of trapezoids    */

    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    local_int = Trap(local_a, local_b, local_n, h);

    if (my_rank != 0)
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
                MPI_COMM_WORLD);
    else {
        total_int = local_int;
        for (source = 1; source < comm_sz; source++) {
            MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_int += local_int;
        }
    }
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n",
```

```
            a, b, total_int);
    }
    MPI_Finalize();
    return 0;
} /*    main    */
```

程序 2 Get_input 函数

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
        int* n_p) {
    int dest;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
        for (dest = 1; dest < comm_sz; dest++) {
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else { /* my_rank != 0 */
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
    }
}   /* Get_input */
```

3 采用 OpenMP 模拟生产者与消费者的消息传递，其中每一个线程有一个共享消息队列，当一个线程要向另外一个线程“发送消息”时，它将消息放入目标线程的消息队列中。一个线程接收消息时只需从它的消息队列的头部取出消息

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "queue_lk.h"//在 queue_lk.h 中已定义入队函数和出队函数 Enqueue 和 Dequeue;
/*void Enqueue(struct queue_s* q_p, int src, int mesg);
int Dequeue(struct queue_s* q_p, int* src_p, int* mesg_p);*/

const int MAX_MSG = 10000;
void Usage(char* prog_name);
void Send_msg(struct queue_s* msg_queues[], int my_rank,
        int thread_count, int msg_number);
void Try_receive(struct queue_s* q_p, int my_rank);
int Done(struct queue_s* q_p, int done_sending, int thread_count);
```

```
int main(int argc, char* argv[]) {
    int thread_count;
    int send_max;
    struct queue_s** msg_queues;
    int done_sending = 0;

    if (argc != 3) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    send_max = strtol(argv[2], NULL, 10);
    if (thread_count <= 0 || send_max < 0) Usage(argv[0]);

    msg_queues = malloc(thread_count*sizeof(struct queue_node_s*));

#   pragma omp parallel num_threads(thread_count) \
        default(none) shared(thread_count, send_max, msg_queues, done_sending)
    {
        int my_rank = omp_get_thread_num();
        int msg_number;
        srandom(my_rank);
        msg_queues[my_rank] = Allocate_queue();
#       pragma omp barrier;//显示路障
        for (msg_number = 0; msg_number < send_max; msg_number++) {
            Send_msg(msg_queues, my_rank, thread_count, msg_number);
            Try_receive(msg_queues[my_rank], my_rank);
        }
#       pragma omp atomic;   //保护临界区
        done_sending++; // 用于判断是否所有线程都已发送完消息的变量
        while (!Done(msg_queues[my_rank], done_sending, thread_count))
            Try_receive(msg_queues[my_rank], my_rank);
        Free_queue(msg_queues[my_rank]);
        free(msg_queues[my_rank]);
    }   /* omp parallel */
    free(msg_queues);
    return 0;
}/* main */
int Done(struct queue_s* q_p, int done_sending, int thread_count) {
    int queue_size = q_p->enqueued - q_p->dequeued;
    if (queue_size == 0 && done_sending == thread_count)
        return 1;
    else
        return 0;
}    /* Done */
```

完成上述代码，并实现函数 Send_msg 和 Try_receive，要求使用锁来保护临界区。

答：Send_msg 和 Try_receive 函数的具体实现如下：

```
void Send_msg(struct queue_s* msg_queues[], int my_rank,
        int thread_count, int msg_number) {
    int mesg = -msg_number;
    int dest = random() % thread_count;
    struct queue_s* q_p = msg_queues[dest];
    omp_set_lock(&q_p->lock);
    Enqueue(q_p, my_rank, mesg);
    omp_unset_lock(&q_p->lock);
}   /* Send_msg */


void Try_receive(struct queue_s* q_p, int my_rank) {
    int src, mesg;
    int queue_size = q_p->enqueued - q_p->dequeued;

    if (queue_size == 0) return;
    else if (queue_size == 1) {
        omp_set_lock(&q_p->lock);
        Dequeue(q_p, &src, &mesg);
        omp_unset_lock(&q_p->lock);
    } else
        Dequeue(q_p, &src, &mesg);
    printf("Thread %d > received %d from %d\n", my_rank, mesg, src);
}    /* Try_receive */
```