OpenMP Codes

RUN : gcc -o filename-fopenmp filename.c

COMPILE : ./filename

Get Information

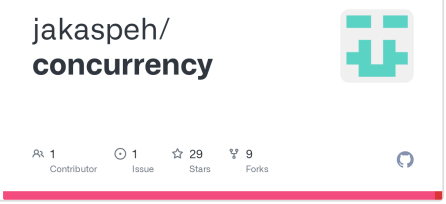
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int nthreads, tid, procs, maxt, inpar, dynamic, nested;

/* Start parallel region */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();

  /* Only master thread does this */
  if (tid == 0)
    {
    printf("Thread %d getting environment info...\n", tid);

    /* Get environment information */
    procs = omp_get_num_procs();
    nthreads = omp_get_num_threads();
```

```
    maxt = omp_get_max_threads();
    inpar = omp_in_parallel();
    dynamic = omp_get_dynamic();
    nested = omp_get_nested();

    /* Print environment information */
    printf("Number of processors = %d\n", procs);
    printf("Number of threads = %d\n", nthreads);
    printf("Max threads = %d\n", maxt);
    printf("In parallel? = %d\n", inpar);
    printf("Dynamic threads enabled? = %d\n", dynamic);
    printf("Nested parallelism supported? = %d\n", nested);

    }

  }  /* Done */

}
```

GitHub - jakaspeh/concurrency

Contribute to jakaspeh/concurrency development by creating an account on GitHub.

https://github.com/jakaspeh/concurrency

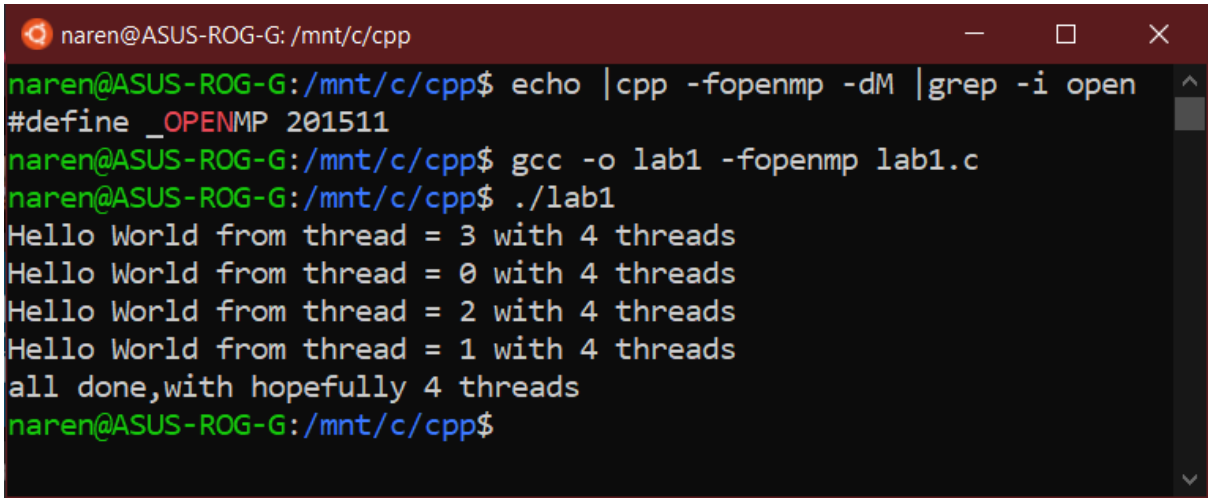


Parallel Loop reduction

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

int main (int argc, char *argv[])
{
int   i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);

printf("   Sum = %f\n",sum);

}
```

Hello World

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int nthreads = 4;
    omp_set_num_threads(nthreads);

    #pragma omp parallel
    {
        //FORK
        int id = omp_get_thread_num();

        printf("Hello World from thread = %d",id);
        printf(" with %d threads\n",omp_get_num_threads());
    }
        //JOIN
    printf("all done,with hopefully %d threads \n",nthreads);
}
```



Hello World (ordered)

```
#include<stdio.h>
#include<omp.h>
int main()
{

    int  proc = omp_get_num_procs();
    printf("\nNo. of processors : %d\n", proc);
    int nthreads,id,i;
    nthreads = 4;
    printf("No. of Threads : %d\n", nthreads);
    omp_set_num_threads(nthreads);
    #pragma omp parallel
    {
        #pragma omp for ordered
        for(i = 0; i < nthreads; i++){
        #pragma omp ordered
        id = omp_get_thread_num();
        printf("\nHello World from thread = %d\n",id);
        }
```

```
    }
}
```

```
naren@ASUS-ROG-G:/mnt/c/cpp$ ./hello_world_ordered
No. of processors : 12
No. of Threads : 4

Hello World from thread = 0

Hello World from thread = 1

Hello World from thread = 2

Hello World from thread = 3
```

Sum of n numbers - Critical Section

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int nthreads = 4;
    omp_set_num_threads(nthreads);
    int partial_Sum, total_Sum ,n;
    printf("\nEnter the value of 'N' : ");
    scanf("%d",&n);
    printf("\n");

    #pragma omp parallel private(partial_Sum) shared(total_Sum,n)
    {
        partial_Sum = 0;
        total_Sum = 0;
        int id = omp_get_thread_num();

        #pragma omp for
            for(int i = 1; i <= n; i++){
                partial_Sum += i;
                printf("Sum from thread %d : %d\n",id,partial_Sum);
            }

        //Create thread safe region.
        #pragma omp critical
        {
              //add each threads partial sum to the total sum
              total_Sum += partial_Sum;
        }
    }
    printf("\nTotal Sum: %d\n", total_Sum);
    return 0;
}

gcc -o sum-fopenmp sum.c
./sum
```

With and Without Critical Section

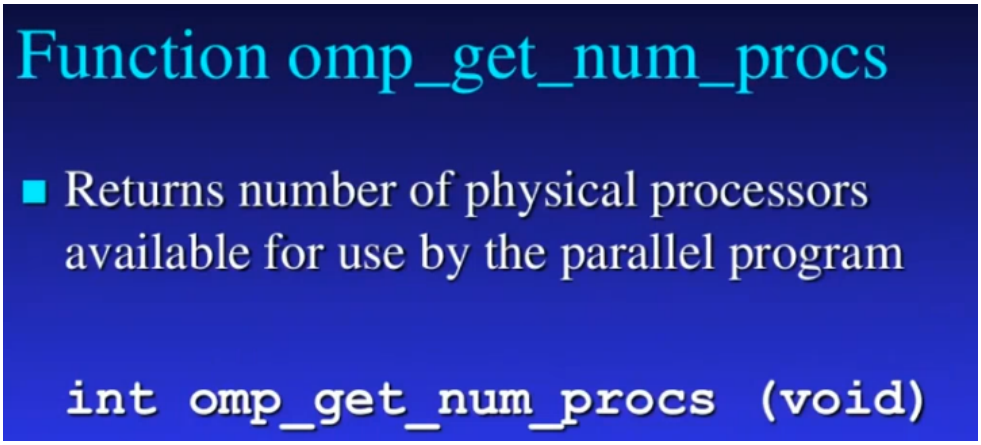
Prime numbers from 1 to n

```
#include<stdio.h>
#include<omp.h>

int main(int argc, char** argv){
    int nthreads = 4;
    omp_set_num_threads(nthreads);

    int n,count,final_count;
    printf("Enter the value of 'n' : ");
    scanf("%d",&n);
    int prime[n];
    printf("Prime nos. 1 to %d:\n",n);

    for(int i=1; i<=n; i++){
        count = 0;
        #pragma omp parallel private(count) shared(n)
        {
        for(int j=2; j<=i/2; j++){
            if(i%j==0){
                count++;
                break;
            }
        }

        if(count==0 && i!=1)
            printf("%d ",i);
    }
}
printf("\n");
return 0;
}
```

No. of Processor = No. of Threads

```
#include<stdio.h>
#include<omp.h>
int main()
{

    int  proc = omp_get_num_procs();
    printf("\nNo. of processors : %d\n", proc);
    int nthreads,id;
    nthreads = proc;
    printf("No. of Threads : %d\n", nthreads);
    omp_set_num_threads(nthreads);
    #pragma omp parallel private(id)
    {

        int id = omp_get_thread_num();

        printf("\nHello World from thread = %d",id);
        printf(" with %d threads\n",omp_get_num_threads());
    }
```

```
    printf("\n*****all done,with hopefully %d threads***** \n\n",nthreads);
}
```



Array Addition

```
#include<stdio.h>
#include<omp.h>
void main()
{
int a[5]={1,2,3,4,5};
int b[5]={6,7,8,9,10};
int c[5];
int tid;

#pragma omp parallel num_threads(5)
{
tid=omp_get_thread_num();
c[tid]=a[tid]+b[tid];
printf("c[%d]=%d\n",tid,c[tid]);
}

}
```

Sections : Allocate different work to different thread : Functional Parallelism

```
#pragma omp parallel sections num_threads(3)
{
   #pragma omp section
     {
       printf("Hello World One");
     }
   #pragma omp section
     {
       printf("Hello World Two");
     }
   #pragma omp section
     {
       printf("Hello World Three");
     }
```

```
}
```

Matrix Addition

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
void main()
{
int tid;
int i,j;
int rows,cols;

printf("Enter Number of Rows of matrices\n");
scanf("%d",&rows);
printf("Enter Number of Columns of matrices\n");
scanf("%d",&cols);

int a[rows][cols];
int b[rows][cols];
int c[rows][cols];

int *d,*e,*f;

printf("Enter %d elements of first matrix\n",rows*cols);
for(i=0;i<rows;i++)
  for(j=0;j<cols;j++)
    {
       scanf("%d",&a[i][j]);
    }

printf("Enter %d elements of second matrix\n",rows*cols);
for(i=0;i<rows;i++)
  for(j=0;j<cols;j++)
    {
       scanf("%d",&b[i][j]);
    }

d=(int *)malloc(sizeof(int)*rows*cols);
e=(int *)malloc(sizeof(int)*rows*cols);
f=(int *)malloc(sizeof(int)*rows*cols);

d=(int *)a;
e=(int *)b;
f=(int *)c;

//Concurrent or parallel matrix addition
#pragma omp parallel num_threads(rows*cols)
  {
    tid=omp_get_thread_num();
    f[tid]=d[tid]+e[tid];
  }

printf("Values of Resultant Matrix C are as follows:\n");

for(i=0;i<rows;i++)
  for(j=0;j<cols;j++)
    {
```

```
        printf("Value of C[%d][%d]=%d\n",i,j,c[i][j]);
    }

}
```

Private - Shared

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int nthreads = 3;
    omp_set_num_threads(nthreads);
    int private_Sum, shared_Sum ,n;
    printf("\nEnter the value of 'N' : ");
    scanf("%d",&n);

    #pragma omp parallel private(private_Sum) shared(shared_Sum)
    {
        private_Sum = 0;
        shared_Sum = 0;
        int id = omp_get_thread_num();

        #pragma omp for
            for(int i = 1; i <= n; i++){
                private_Sum += i;
                shared_Sum += i;

            }
    }
    printf("\nShared Sum : %d",shared_Sum);
    printf("\nPrivate Sum : %d\n",private_Sum);


    return 0;
}
```

Sequential Prefix Sum

```
#include<stdio.h>
#include<math.h>
int highestPowerof2(int n){
    int res = 0,i;
    for (i=n; i>=1; i--)
    {
        // If i is a power of 2
        if ((i & (i-1)) == 0)
        {
            res = i;
            break;
        }
    }
    return res;
}
int main(int argc, char** argv){
  int arr[]={7,8,16,17,54,62,73,77,82,88,92,97};
```

```
  int n=sizeof(arr)/sizeof(arr[0]);//size of array
  int output[n],i=1,j;
  int x=(int)highestPowerof2(n);//to find the nearest power of 2
    int y= (int)log2(x);// finding the exponent of the nearest power


  printf("n=%d",n);
    printf("\nx=%d",x);
    printf("\ny=%d\n",y);
  int count=1;
    printf("Input given:  ");
  for(i=0;i<n;i++){
  printf("%d  ",arr[i]);
  }
  printf("\n");


  output[0]=arr[0];// initial value remains the same

  for(i=1;i<=x;i=i*2){
    // distance increases by power of 2 for every iteration
    for(j=i;j<n;j++){
      output[j]=arr[j]+arr[j-i];
    }
    printf("I=%d, dist=%d:  ",count,i);
    for(j=0;j<n;j++){
      //using the array arr[] as a temporary array to store the sum
      arr[j]=output[j];
      printf("%d  ",arr[j]);
    }
    printf("\n");
    count++;
    }


  printf("\nThe measurements to cut sandwiches are:");
  #pragma omp for
  for(i=0;i<n;i++)
  {
  printf(" %d  ",output[i]);
}
printf("\n");
}
```

Quick Sort

```
#include<stdio.h>
#include<omp.h>

int k=0;

int partition(int arr[], int low_index, int high_index)
{
int i, j, temp, key;
key = arr[low_index];
i= low_index + 1;
j= high_index;
while(1)
{
```

```
while(i < high_index && key >= arr[i])
    i++;
while(key < arr[j])
    j--;
if(i < j)
{
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
else
{
temp= arr[low_index];
arr[low_index] = arr[j];
arr[j]= temp;
return(j);
}
}
}


void quicksort(int arr[], int low_index, int high_index)
{
int j;

if(low_index < high_index)
{
j = partition(arr, low_index, high_index);
printf("Pivot element with index %d has been found out by thread %d\n",j,k);

#pragma omp parallel sections
{
    #pragma omp section
    {
        k=k+1;
        quicksort(arr, low_index, j - 1);
    }

    #pragma omp section
    {
        k=k+1;
        quicksort(arr, j + 1, high_index);
    }

}
}
}


int main()
{
int arr[100];
int n,i;

printf("Enter the value of n\n");
scanf("%d",&n);
printf("Enter the %d number of elements \n",n);

for(i=0;i<n;i++)
{
scanf("%d",&arr[i]);
}
```

```
quicksort(arr, 0, n - 1);

printf("Elements of array after sorting \n");

for(i=0;i<n;i++)
{
printf("%d\t",arr[i]);
}

printf("\n");
}
```

Merge Sort

```
#include<stdio.h>
#include<omp.h>

void merge(int array[],int low,int mid,int high)
{
  int temp[30];
  int i,j,k,m;
  j=low;
  m=mid+1;
  for(i=low; j<=mid && m<=high ; i++)
  {
     if(array[j]<=array[m])
     {
         temp[i]=array[j];
         j++;
     }
     else
     {
         temp[i]=array[m];
         m++;
     }
  }
  if(j>mid)
  {
     for(k=m; k<=high; k++)
     {
         temp[i]=array[k];
         i++;
     }
  }
  else
  {
     for(k=j; k<=mid; k++)
     {
        temp[i]=array[k];
        i++;
     }
  }
  for(k=low; k<=high; k++)
     array[k]=temp[k];
}


void mergesort(int array[],int low,int high)
{
```

```
 int mid;
 if(low<high)
 {
   mid=(low+high)/2;

   #pragma omp parallel sections num_threads(2)
    {
      #pragma omp section
        {
          mergesort(array,low,mid);
        }

      #pragma omp section
        {
          mergesort(array,mid+1,high);
        }
    }
   merge(array,low,mid,high);
 }
}


int main()
{
 int array[50];
 int i,size;
 printf("Enter total no. of elements:\n");
 scanf("%d",&size);
 printf("Enter %d elements:\n",size);
 for(i=0; i<size; i++)
 {
   scanf("%d",&array[i]);
 }
 mergesort(array,0,size-1);
 printf("Sorted Elements as follows:\n");
 for(i=0; i<size; i++)
    printf("%d ",array[i]);
 printf("\n");
 return 0;
}
```

Deadlock Problem - LOCKS

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb)

  #pragma omp sections nowait

    {
    // PROCESS 1

    #pragma omp section
      {

    // Setting Locks
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        // Set A
        a[i] = ..
```

```
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        // Update B
        b[i] = .. a[i] ..

    //Unsetting Locks
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);

      }

    //PROCESS 2

    #pragma omp section
      {
      // Setting Locks
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
      // Set B
        b[i] = ...

      omp_set_lock(&locka);
      for (i=0; i<N; i++)

        // Update A
        a[i] = .. b[i] ..

      //Unsetting Locks
      omp_unset_lock(&locka);
      omp_unset_lock(&lockb);
      }
    }  /* end of sections */
  }  /* end of parallel region */
```

Linked List

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

#ifndef N
#define N 5
#endif
#ifndef FS
#define FS 38
#endif

struct node {
   int data;
   int fibdata;
   struct node* next;
};

int fib(int n) {
   int x, y;
   if (n < 2) {
      return (n);
   } else {
      x = fib(n - 1);
```

```
      y = fib(n - 2);
    return (x + y);
   }
}

void processwork(struct node* p)
{
   int n;
   n = p->data;
   p->fibdata = fib(n);
}

struct node* init_list(struct node* p) {
    int i;
    struct node* head = NULL;
    struct node* temp = NULL;

    head = malloc(sizeof(struct node));
    p = head;
    p->data = FS;
    p->fibdata = 0;
    for (i=0; i< N; i++) {
       temp  =  malloc(sizeof(struct node));
       p->next = temp;
       p = temp;
       p->data = FS + i + 1;
       p->fibdata = i+1;
    }
    p->next = NULL;
    return head;
}

int main(int argc, char *argv[]) {
     double start, end;
     struct node *p=NULL;
     struct node *temp=NULL;
     struct node *head=NULL;

   printf("Process linked list\n");
     printf("  Each linked list node will be processed by function 'processwork()'\n");
     printf("  Each ll node will compute %d fibonacci numbers beginning with %d\n",N,FS);

     p = init_list(p);
     head = p;

     start = omp_get_wtime();
     {
        while (p != NULL) {
       processwork(p);
       p = p->next;
        }
     }

     end = omp_get_wtime();
     p = head;
   while (p != NULL) {
        printf("%d : %d\n",p->data, p->fibdata);
        temp = p->next;
        free (p);
        p = temp;
     }
   free (p);
```

```
     printf("Compute Time: %f seconds\n", end - start);

     return 0;
}
```

Matrix Multiplication

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

#define NRA 62                  /* number of rows in matrix A */
#define NCA 15                  /* number of columns in matrix A */
#define NCB 7                   /* number of columns in matrix B */

int main (int argc, char *argv[])
{
int tid, nthreads, i, j, k, chunk;
double  a[NRA][NCA],           /* matrix A to be multiplied */
  b[NCA][NCB],           /* matrix B to be multiplied */
  c[NRA][NCB];           /* result matrix C */

chunk = 10;                    /* set loop iteration chunk size */

/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    printf("Initializing matrices...\n");
    }
  /*** Initialize matrices ***/
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    {
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
```

```
  }   /*** End of parallel region ***/

/*** Print results ***/
printf("******************************************************\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
  {
  for (j=0; j<NCB; j++)
    printf("%6.2f   ", c[i][j]);
  printf("\n");
  }
printf("******************************************************\n");
printf ("Done.\n");

}
```

▼ OpenMP Mat Multiplication

```
#include <omp.h>
#include <sys/time.h>


#define N 1000

int A[N][N];
int B[N][N];
int C[N][N];

int main()
{
    int i,j,k;
    struct timeval tv1, tv2;
    struct timezone tz;
  double elapsed;
    omp_set_num_threads(omp_get_num_procs());
    for (i= 0; i< N; i++)
        for (j= 0; j< N; j++)
  {
            A[i][j] = 2;
            B[i][j] = 2;
  }
    gettimeofday(&tv1, &tz);
    #pragma omp parallel for private(i,j,k) shared(A,B,C)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }


    gettimeofday(&tv2, &tz);
    elapsed = (double) (tv2.tv_sec-tv1.tv_sec) + (double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
    printf("elapsed time = %f seconds.\n", elapsed);

    /*for (i= 0; i< N; i++)
    {
        for (j= 0; j< N; j++)
        {
```

```
            printf("%d\t",C[i][j]);
        }
        printf("\n");
    }*/
}
```

Work Sharing - Sections

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     50

int main (int argc, char *argv[])
{
int i, nthreads, tid;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i<N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  c[i] = d[i] = 0.0;
  }

#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d doing section 1\n",tid);
      for (i=0; i<N; i++)
        {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
      }

    #pragma omp section
      {
      printf("Thread %d doing section 2\n",tid);
      for (i=0; i<N; i++)
        {
        d[i] = a[i] * b[i];
        printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
      }

    }  /* end of sections */
```

```
    printf("Thread %d done.\n",tid);

  }  /* end of parallel section */

}
```