PRACTICAL - 01

AIM: Introduction toFork() system call and Wait() function with example.

Program:

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

int main(int argc,char *argv[])
{
        int iden;
        iden=fork();
        if(iden==0)
        {
                printf("this is from child:child id is:%dand parent id is %d\n",getpid(),getppid());
        }
        if(iden>0)
        {
                printf("this is from parent:child id:%d and parent id is %d\n",getpid(),getppid());
        }
        wait();
        return 0;
}
```

To compile the program:

```
$>gccforktest.c –o forkwait
```

To run the program:

```
$> ./forkwait
```

Output:

```
this is from child:child id is:5011and parent id is 5010
this is from parent:child id:5010 and parent id is 4824
```

PRACTICAL - 02

AIM: Implementation of a process tree.

Program:

```
#include<stdio.h>
int main(int argc,char *argv[])
{         int iden,iden1;
          iden=fork();
          if(iden==0)
          {    printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
             iden1=fork();
                    if(iden1==0)
                    {         printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
                     }
          }
          if(iden>0)
       {          printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
                    iden=fork();
                    if(iden==0)
                 {
          printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
                              iden1=fork();
          if(iden1==0)
                       {    printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
             }
                     }
                    if(iden>0)
                    {         iden=fork();
                              if(iden==0)
                              {      printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
                                     iden1=fork();
          if(iden1==0)
                     {     printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
                    }
           }
                              if(iden>0)
                              {         iden=fork();
                                        if(iden==0)
                                        {
                                                  printf("child id is:%d and parent id
                                                  is:%d\n",getpid(),getppid());
                                        iden1=fork();
                              if(iden1==0)
                          {      printf(" child id is:%d and parent id is:%d\n",getpid(),getppid());
                           }
                                        }
                              }
                    }
          }
wait();
          return 0;
}
```

Output:

```
child id is:5212 and parent id is:5211
child id is:5214 and parent id is:5212
child id is:5212 and parent id is:5211
child id is:5211 and parent id is:4824
child id is:5211 and parent id is:4824
child id is:5213 and parent id is:5211
child id is:5211 and parent id is:4824
child id is:5215 and parent id is:5211
child id is:5218 and parent id is:5215
child id is:5211 and parent id is:4824
child id is:5216 and parent id is:5211
child id is:5211 and parent id is:4824
child id is:5215 and parent id is:5211
```

PRACTICAL - 03

AIM: Implement a parallel program to calculate sum and average of 1 to N numbers using shared memory.

Program to calculate sum of 1 to N numbers:

```
#include<stdio.h>
#include<sys/ipc.h>
#include<stdlib.h>
#include<sys/shm.h>
Void busy_wait()
{
        int number1, number 2;
        int A;
        for(number1=0; number1<100*10; number1++)
        {
                for(number2=0; number2<80000; number2++)
                {
                        A= number1* number2;
                }
        }
        return;
}
int main()
{
        int piden1,to1;
        int *buffer1, number1;
        int iden;
#ifdef _PSEQ1
        to1=0;
        for(number1=1; number1<=100; number1++)
        {
                to1+= number1;
                busy_wait();
        }
        printf("%d\n",to1);
#endif
#ifdef _PPARL1
        if((iden=shmget(IPC_PRIVATE,sizeof(int)*2,IPC_CREAT|0777))<0)
        {
                perror("Error in getmemory shared\n");
                exit(1);
        }
        buffer1=(int*)shmat(iden,NULL,0);
        piden1=fork();
        if(piden1>0)
        {
                buffer1[0]=0;
                for(number1=1; number1<=50; number1++)
                {
                        buffer1[0]+= number1;
                        busy_wait();
                }
                wait(0);
                printf("%d\n",buffer1[0]+buffer1[1]);
        }
        else if(piden1<0)
        {
                perror("error::\n");
        }
        else
        {
                buffer1[1]=0;
                for(number1=51; number1<=100; number1++)
                {
                        buffer1[1]+= number1;
                        busy_wait();
                }
        }
#endif
        return 0;
}
```

To Compile the program in sequential form:

```
$>gcc –D_PSEQ1 expl1.c –o pseq1
```

To run the program:

```
$>time  ./pseq1
```

The time taken to generate the results is ≈ 17secs

To Compile the program in parallel form:

```
$>gcc –D_PPAR1 expl1.c –o pseq1
```

To run the program:

```
$>time  ./pseq1
```

The time taken to generate the results is ≈ 8 secs

Output:

```
5050
```

Program to calculate average of 1 to N numbers:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/shm.h>
#include<sys/ipc.h>
#include<sys/types.h>

voidfunc()
{
        int io,jo;
        for(io=1;io<80;io++)
        {
                for(jo=1;jo<400;jo++)
                {
                        io*jo;
                }
        }
}

int main()
{
        const int number =100;
        int Add1=0,Add2=0;
        int *buffer;
        int io,po1,fd;
        float avrg;

#ifdef _NONPARR

        for(io=0;io<= numbe;io++)
        {
                func();
                Add1+=io;
        }

        printf("Final sum is:%d\n",Add1);
        avrg=(float)Add1/ number;
        printf("Average is: %f\n\n",avrg);

#endif
#ifdef _PARR

        if((fd=shmget(IPC_PRIVATE,sizeof(int)*2,IPC_CREAT|0777))<0)
```

```
	{
		perror("Error\n");
		exit(1);
	}

	if((buffer=(int*)shmat(fd,NULL,0))==(void*)-1)
	{
		perror("Error1\n");
		exit(1);
	}

	po1=fork();
	if(po1>0)
	{
		for(io=1;io<= number/2;io++)
		{
			func();
			Add1+=io;
		}

		buffer[0]=Add1;
		printf("Sum by parent: %d\n",buffer[0]);
		wait(0);
		printf("Total sum: %d\n",(buffer[0]+buffer[1]));
		avrg=(float)(buffer[0]+buffer[1])/n;
		printf("Average is: %f\n\n",avrg);
	}
	else if(po1==0)
	{
		for(io=number/2+1;io<=numbe;io++)
		{
			func();
			Add2+=io;
		}

		buffer[1]=Add2;
		printf("Sum by child: %d\n",buffer[1]);
	}
	else
	{
		printf("Erroneous code\n");
	}
	#endif

	return 0;
}
```

Output:

```
[exam2@LINTEL exam2]$ gccavg.c -D_NONPARR -o t1
[exam2@LINTEL exam2]$ gccavg.c -D_PARR -o t2
[exam2@LINTEL exam2]$ time ./t1
Final sum is:5050
Average is: 50.500000


real    0m0.735s
user    0m0.740s
sys     0m0.000s
[exam2@LINTEL exam2]$ time ./t2
Sum by parent: 1275
Sum by child: 3775
Total sum: 5050
Average is: 50.500000


real    0m0.664s
user    0m0.670s
sys     0m0.000s
[exam2@LINTEL exam2]$
```

PRACTICAL - 04

AIM: Implement a parallel program to find minimum of numbers using shared memory.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/shm.h>
#include<sys/ipc.h>
#include<sys/types.h>
voidfunc()
{
        int io,jo;
        for(io=1;io<80000;io++)
        {
                for(jo=1;jo<800;jo++)
                {io*jo;}
        }
}
int main()
{
        const int nAn=100;
        int Add1=0,Add2=0;
        int *buffer;
        int  inpt1[nAn],prmin1;
        int io,io1,po1,fd;
        float avrg;
        for(io1=0;io1<nAn;io1++)
        {
                inpt1[io1]=io1+1;
        }
        for(io1=0;io1<nAn;io1++)
        {
                printf("%d::%d\n",io1,inpt1[io1]);
        }


#ifdef _NONPARR
        prmin1=inpt1[0];
        for(io1=1;io1<=nAn;io1++)
        {
                func();
```

```
				if(prmin1>inpt1[io1])
					prmin1=inpt1[io1];
		}
		printf("Result is: %d\n",prmin1);

#endif

#ifdef _PARR

		if((fd=shmget(IPC_PRIVATE,sizeof(int)*2,IPC_CREAT|0777))<0)
		{
				perror("Error\n");
				exit(1);
		}
		if((buffer=(int*)shmat(fd,NULL,0))==(void*)-1)
		{
				perror("Error1\n");
				exit(1);
		}

		po1=fork();

		if(po1>0)
		{
				prmin1=inpt1[0];
				for(io1=1;io1<=nAn/2;io1++)
				{
					func();
					if(prmin1>inpt1[io1])
						prmin1=inpt1[io1];
				}
				printf("Result is: %d %d\n",getpid(),prmin1);
				buffer[0]=prmin1;
				wait(0);
				printf("Result is: %d\n\n",(buffer[0]<buffer[1])?buffer[0]:buffer[1]);
		}
		else if(po1==0)
		{
				prmin1=inpt1[nAn/2+1];
				for(io1=nAn/2+2;io1<=nAn;io1++)
				{
					func();
					if(prmin1>inpt1[i1])
						prmin1=inpt1[io1];
				}
				printf("Result is: %d %d\n",getpid(),prmin1);
		buffer[1]=prmin1;
		printf("Sum by child: %d\n",buffer[1]);
	}
	else
	{
		printf("Erroneous code\n");
	}
#endif
	return 0;
}
```

Output:

```
Result is: 5238 1
Result is: 5239 52
Sum by child: 52
Result is: 1


real    0m9.843s
user    0m18.360s
sys     0m0.000s
[exam2@LINTEL exam2]$
```

PRACTICAL - 05

AIM: Implement a parallel program to perform matrix multiplication using shared memory.

Program:

explmatrix1.c:

```
#include<stdio.h>
#include<sys/ipc.h>
#include<stdlib.h>
#include<sys/shm.h>
void busyfunc1()
{
	int io1,io2;
	int ioi;
	for(io1=0;io1<100;io1++)
		for(io2=0;io2<100;io2++)
			ioi=io1*io2;
}
int main(int argc,char* argv[])
{
	const int XRW=8,XCL=8;
	const int YRW=8,YCL=8;
	inti io,iod;
	int NoOfProc=4,priden;
	int rro1,cco1;
	int rro2,cco2;
	int to1,to2;
	int size=XRW/NoOfProc;
	int *xAx,*yAy;
	int *zAz;
	iden=shmget(IPC_PRIVATE,sizeof(int)*(XRW*XCL),0777|IPC_CREAT);
	xAx=(int*)shmat(iden,NULL,0);
	iden=shmget(IPC_PRIVATE,sizeof(int)*(YRW*YCL),0777|IPC_CREAT);
	yAy=(int*)shmat(iden,NULL,0);
	iden=shmget(IPC_PRIVATE,sizeof(int)*(XRW*YCL),0777|IPC_CREAT);
	zAz=(int*)shmat(iden,NULL,0);
	for(rro1=0;rro1<XRW;rro1++)
	{
		for(cco1=0;cco1<YCL;cco1++)
		{
			xAx[rro1*XCL+cco1]=1;
		}
	}
	for(rro1=0;rro1<XRW;rro1++)
	{
		for(cco1=0;cco1<YCL;cco1++)
		{
			yAy[rro1*YCL+cco1]=1;
		}
	}
	/*
	for(rro1=0;rro1<XRW;rro1++)
	{
		for(cco1=0;cco1<XCL;cco1++)
		{
			printf("%d ",xAx[rro1*XCL+cco1]);
		}
		printf("\n");
	}
	for(rro1=0;rro1<YRW;rro1++)
	{
		for(cco1=0;cco1<YCL;cco1++)
		{
			printf("%d ",yAy[rro1*YCL+cco1]);
		}
		printf("\n");
	}
	*/
	proc_func1(NoOfProc,&priden);
	to1=size*priden;
	to2=to1+size-1;
	for(rro1=to1;rro1<=to2;rr1++)
	{
		for(cco1=0;cco1<YCL;cco1++)
		{
			zAz[rro1*YCL+cco1]=0;
			for(ioi=0;ioi<XCL;ioi++)
			{
				busyfunc1();

	zAz[rro1*YCL+cco1]+=xAx[rro1*XCL+ioi]*yAy[ioi*YCL+cco1];
			}
		}
	}
	if(priden==0)
	{
		Int pop=0;
		for(ioi=0;ioi<NoOfProc;ioi++)
		{
			wait(0);
		}
		for(rro1=0;rro1<XRW;rro1++)
		{
			for(cco1=0;cco1<YCL;cco1++)
			{
				printf("%d ",zAz[rro1*YCL+cco1]);
			}
			printf("\n");
		}
	}
	return 0;
}
```

proc.c:

```
#include<stdio.h>
#include<stdlib.h>
void proc_func1(intno_of_proc,int *prid)
{
	int piden1,po1;
	int ioi,prcount=0;
	/*
	piden1 = fork();
	if(piden1>0)
	{
		printf("process:::%d %d\n",getpid1(),getppid1());
	*/
	*priden=0;
		for(po1=1;po1<NoOfProc;po1++)
		{
			if(*priden==0)
			{
			piden1=fork();prcount++;
			}
			if(piden1>0)
			{
				continue;
			}
			else if(piden1==0&&*priden==0)
			{
				*priden=po1;
			}
			else
```

```
                    {
                    }
            }
    return;
}
#ifdef _PROCMAIN1
int main(int argc,char* argv[])
{
    int ioi;
    int NoOfProc=5,priden;
    proc_func1(noOfProc,&priden);
    for(ioi=0;ioi<NoOfProc;ioi++)
    {
            if(ioi==priden&&priden>0)
            {
                    sleep(2);
                    printf("ioi::%d priden::%d\n",ioi,priden);
            }
    }
    if(priden==0)
    {
            for(ioi=0;ioi<NoOfProc;ioi++)
            {
                    wait(0);
            }
    }
    return 0;
}
#endif
```

Output:

```
8 8 8 8
8 8 8 8
8 8 8 8
8 8 8 8
```

PRACTICAL - 06

AIM:Implement a program to demonstrate a pthread_create() and pthread_join() functions in pthread library.

Program:

```
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

void*thread_function(void*);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  cnter = 0;

main()
{
        pthread_tthread_id[NTHREADS];
        int io, jo;

        for(io=0; io< NTHREADS; io++)
        {
                pthread_create(&thread_id[io], NULL, thread_function, NULL );
        }

        for(jo=0; jo < NTHREADS; jo++)
        {
                pthread_join(thread_id[jo], NULL);
        }

        printf("Final counter value: %d\n", cnter);
}

void*thread_function(void*dummyPtr)
{
         printf("Thread number %ld\n", pthread_self());
        pthread_mutex_lock(&mutex1 );
        cnter++;
        pthread_mutex_unlock(&mutex1 );
}
```

To Compile the program :

```
$>gcc join1.c –o threadsync -lpthread
```

To run the program:

```
$>./threadsync
```

Output:

```
Thread number 1026
Thread number 2051
Thread number 3076
Thread number 4101
Thread number 5126
Thread number 6151
Thread number 7176
Thread number 8201
Thread number 9226
Thread number 10251
Final counter value: 10
```

PRACTICAL - 07

AIM:Introduction to mutual exclusion mechanism in pthread library.

Program - 01:

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

voidfunc()
{
        int io,jo;
        for(io=1;io<80;io++)
        {
                for(jo=1;jo<800;jo++)
                {
                        io*jo;
                }
        }
}

int xcnt1=0;
pthread_mutex_t po1=PTHREAD_MUTEX_INITIALIZER;

void* func1(void* to1)
{
        int xox;
        int io1;
        for(io1=1; io1<=500; io1++)
        {


                pthread_mutex_lock(&po1);
                xox=xcnt1;
                xox++;
                func();
                xcnt1=xox;
                pthread_mutex_unlock(&po1);



        printf("NO. NEVER. NOT::%d\n",xcnt1);
        }
}

int main()
        {
                pthread_t to1,to2;
                pthread_mutex_init(&po1,NULL);
                pthread_create(&to1,NULL,func1,NULL);
                pthread_create(&to2,NULL,func1,NULL);
                pthread_join(to1,NULL);
                pthread_join(to2,NULL);
                return 0;
        }
```

Output:

```
[exam2@LINTEL exam2]$ gccme.c -o t1 -lpthread
[exam2@LINTEL exam2]$ time ./t1
NO. NEVER. NOT::500
NO. NEVER. NOT::1000

real    0m0.138s
user    0m0.140s
sys     0m0.000s
[exam2@LINTEL exam2]$
```

Program – 02:

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

voidfunc()
{
int io,jo;
for(io=1;io<80;io++)
      {
         for(jo=1;jo<800;jo++)
            {
                io*jo;
            }
      }
}
```

```
int xcnt1=0;
pthread_mutex_t po1=PTHREAD_MUTEX_INITIALIZER;

void* func1(void* to1)
{
int xox;
int io1;
for(io1=1; io1<=500; io1++)
      {
         xox=xcnt1;
         xox++;
         func();
         xcnt1=xx;
      }
printf("NO. NEVER. NOT::%d\n",xcnt1);

}

int main()
{
         pthread_t to1,to2;
         pthread_mutex_init(&po1,NULL);
         pthread_create(&to1,NULL,func1,NULL);
         pthread_create(&to2,NULL,func1,NULL);
         pthread_join(to1,NULL);
         pthread_join(to2,NULL);
         return 0;
}
```

Output:

```
[exam2@LINTEL exam2]$ gccme.c -o t1 -lpthread
[exam2@LINTEL exam2]$ time ./t1
NO. NEVER. NOT::500
NO. NEVER. NOT::668

real    0m0.134s
user    0m0.130s
sys     0m0.000s
[exam2@LINTEL exam2]$
```

PRACTICAL - 08

AIM: Implementation of Producer-Consumer problem using pthread library.

Program:

```
# include <stdio.h>
# include <pthread.h>
# define BufferSize 10

void *Producer();
void *Consumer();

int BuffIndx=0;
char *BUFF;

pthread_cond_tBuffer_Not_Full=PTHREAD_COND_INITIALIZER;
pthread_cond_tBuffer_Not_Empty=PTHREAD_COND_INITIALIZER;
pthread_mutex_tmVar=PTHREAD_MUTEX_INITIALIZER;

int main()
{
pthread_toptid,cotid;

   BUFF=(char *) malloc(sizeof(char) * BufferSize);

pthread_create(&potid,NULL,Producer,NULL);
pthread_create(&cotid,NULL,Consumer,NULL);

pthread_join(potid,NULL);
pthread_join(cotid,NULL);


return 0;
}

void *Producer()
{
for(;;)
   {
pthread_mutex_lock(&mVar);
if(BuffIndx==BufferSize)
       {
pthread_cond_wait(&Buffer_Not_Full,&mVar);
```

```
        }
BUFF[BuffIndx++]='@';
printf("Produce : %d \n",BuffIndx);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Not_Empty);
    }

}

void *Consumer()
{
for(;;)
    {
pthread_mutex_lock(&mVar);
if(BuffIndx==-1)
        {
pthread_cond_wait(&Buffer_Not_Empty,&mVar);
        }
printf("Consume : %d \n",BuffIndx--);
pthread_mutex_unlock(&mVar);
pthread_cond_signal(&Buffer_Not_Full);
    }
}
```

Output:

```
[tarang@localhost pp-tw4]$ cc -o Prog08 -lpthread Prog08.c
[tarang@localhost pp-tw4]$ ./Prog08
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Produce : 5
Produce : 6
Produce : 7
Produce : 8
Produce : 9
Produce : 10
Consume : 10
Consume : 9
Consume : 8
Consume : 7
Consume : 6
Consume : 5
Consume : 4
Consume : 3
Consume : 2
Consume : 1
Consume : 0
Produce : 0
Produce : 1
Produce : 2
Produce : 3
Produce : 4
Produce : 5
Produce : 6
Produce : 7
Produce : 8
Produce : 9
Produce : 10
Consume : 10
Consume : 9
Consume : 8
Consume : 7
Consume : 6
Consume : 5
Consume : 4
Consume : 3
Consume : 2
Consume : 1
Consume : 0
Produce : 0
Produce : 1
```

PRACTICAL - 09

AIM: Introduction to MPI.

Program:

```
#include <stdio.h>
#include "mpi.h"
int main( intargc, char *argv[] )
{
int ro1;
int to1;
int Buffo1[1];
MPI_Status po1;
MPI_Init(0,0);
MPI_Comm_rank(MPI_COMM_WORLD,&r1);
MPI_Comm_size(MPI_COMM_WORLD,&t1);
printf( " Heyy I'm %d of %d\n",ro1,to1 );
if(ro1==0)
   {
         printf( " Heyy I'm %d of %d\n",ro1,to1 );
         Buffo1[0]=1;
         MPI_Send(Buffo1,1,MPI_INT,1,420,MPI_COMM_WORLD);
         MPI_Recv(Buffo1,10,MPI_INT,1,420,MPI_COMM_WORLD,&po1);
         printf("recived %d from slave\n",Buffo1[0]);
   }
else
   {
         MPI_Recv(Buffo1,10,MPI_INT,0,420,MPI_COMM_WORLD,&po1);
         printf( "Heyy I'm slave %d of %d\n", ro1, to1 );
         printf("slave recived %d\n",Buffo1[0]);
         Buffo1[0]=20;
         MPI_Send(Buffo1,1,MPI_INT,0,420,MPI_COMM_WORLD);
   }
MPI_Finalize();
return 0;
}
```

Output:

```
Heyy I'm 0 of 2
Heyy I'm 0 of 2
Heyy I'm 1 of 2
Heyy I'm slave 1 of 2
slaverecived 1
recived 20 from slave
```

PRACTICAL - 10

AIM: Implement a program to pass a message in ring using MPI.

Program:

```
#include <stdio.h>
#include "mpi.h"
int main( int argc, char *argv[] )
{
int ro1;
int to1,io1;
int Buffo1[1];
MPI_Status po1;
MPI_Init(0,0);
MPI_Comm_rank(MPI_COMM_WORLD,&ro1);
MPI_Comm_size(MPI_COMM_WORLD,&to1);
if(ro1==0)
   {
          printf( " Heyy I'm %d of %d\n",ro1,to1 );
          Buffo1[0]=1;
          MPI_Send(Buffo1,1,MPI_INT,1,420,MPI_COMM_WORLD);
          MPI_Recv(Buffo1,10,MPI_INT,t1-1,420,MPI_COMM_WORLD,&po1);
          printf("recived %d from slave %d\n",Buffo1[0],to1-1);
   }
else
   {
          MPI_Recv(Buffo1,10,MPI_INT,ro1-1,420,MPI_COMM_WORLD,&po1);
          printf("slave %d recived %d\n",ro1,buff1[0]);
          Buffo1[0]=ro1;
          MPI_Send(Buffo1,1,MPI_INT,(ro1+1)%to1,420,MPI_COMM_WORLD);
   }
MPI_Finalize();
return 0;
}
```

Output:

```
Heyy I'm 0 of 5
slave 1 recived 1
slave 2 recived 1
slave 3 recived 2
slave 4 recived 3
recived 4 from slave 4
```

BEYOND SYLLABUS - 01

AIM: Case study of Open Multi-processing.

OpenMP(Open Multi-Processing) is anAPIthat supports multi-platformshared memorymultiprocessingprogramming inC,C++, andFortran, on mostprocessor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, andWindows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP is managed by the nonprofit technology consortium *OpenMP Architecture Review Board* (or *OpenMP ARB*), jointly defined by a group of major computer hardware and software vendors, includingAMD,IBM,Intel,Cray,HP, Fujitsu, Nvidia, NEC, Microsoft,Texas Instruments, Oracle Corporation, and more. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

An application built with the hybrid model of parallel programming can run on computer cluster using both OpenMP and Message Passing Interface (MPI), or more transparently through the use of OpenMP extensions for non-shared memory systems.

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master *thread* (a series of instructions executed consecutively) *forks* a specified number of slave *threads* and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed Each thread has an *id*attached to it which can be obtained using a function (called omp_get_thread_num()). The thread id is an integer, and the master thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The OpenMP functions are included in a header file labelled *omp.h* in C/C++.

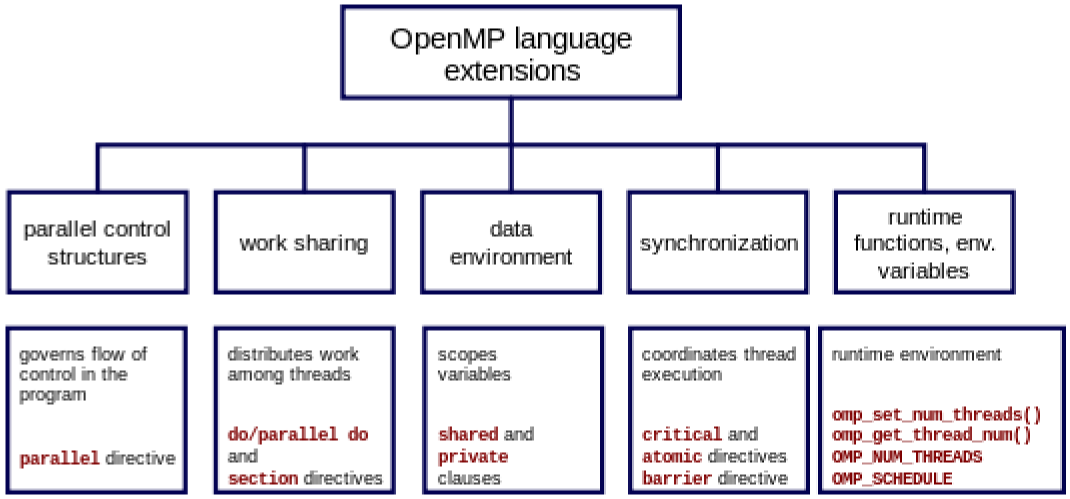
OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Figure - The Core Elements

The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.In C/C++, OpenMP uses #pragmas. The OpenMP specific pragmas are listed below.

Thread creation

The pragma *omp parallel* is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as *master thread* with thread ID 0.

Example (C program): Display "Hello, world." using multiple threads.

```
#include <stdio.h>

int main(void)
{
#pragma omp parallel
printf("Hello, world.\n");
return0;
}
```

Use flag -fopenmp to compile using GCC:

```
$gcc-fopenmphello.c-o hello
```

Output on a computer with two cores, and thus two threads:

```
Hello, world.
Hello, world.
```

However, the output may also be garbled because of the race condition caused from the two threads sharing the standard output.

```
Hello, wHello, woorld.
rld.
```

Work-sharing constructs

Used to specify how to assign independent work to one or all of the threads.

- ***omp for*** **or** ***omp do*****:** used to split up loop iterations among the threads, also called loop constructs.
- ***sections*****:** assigning consecutive but independent code blocks to different threads
- ***single*****:** specifying a code block that is executed by only one thread, a barrier is implied in the end.
- ***master*****:** similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

Example: initialize the value of a large array in parallel, using each thread to do part of the work

```
int main(intargc,char*argv[]){
constint N =100000;
inti, a[N];

#pragma omp parallel for
for(i=0;i<N;i++)
a[i]=2*i;

return0;
}
```

OpenMP clauses

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as *data sharing attribute clauses* by appending them to the OpenMP directive. The different types of clauses are

Data sharing attribute clauses

- ***shared*****:** the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- ***private*****:** the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not

initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.

- ***default*:** allows the programmer to state that the default data scoping within a parallel region will be either *shared*, or *none* for C/C++, or *shared*, *firstprivate*, *private*, or *none* for Fortran. The *none* option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
- ***firstprivate*:** like *private* except initialized to original value.
- ***lastprivate*:** like *private* except original value is updated after construct.
- ***reduction*:** a safe way of joining work from all threads after construct.

Synchronization clauses

- ***critical*:** the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data fromrace conditions.
- ***atomic*:** the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.
- ***ordered*:** the structured block is executed in the order in which iterations would be executed in a sequential loop
- ***barrier*:** each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- ***nowait*:** specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

Scheduling clauses

- ***schedule****(type, chunk)*: This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:

1. ***static*:** Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter *chunk* will allocate chunk number of contiguous iterations to a particular thread.
2. ***dynamic*:** Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter *chunk* defines the number of contiguous iterations that are allocated to a thread at a time.
3. ***guided*:** A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter *chunk*

IF control

- ***if*:** This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.

Initialization

- ***firstprivate*:** the data is private to each thread, but initialized using the value of the variable using the same name from the master thread.
- ***lastprivate*:** the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop. A variable can be both *firstprivate* and *lastprivate*.
- ***threadprivate*:** The data is a global data, but it is private in each parallel region during the runtime. The difference between *threadprivate* and *private* is the global scope associated with threadprivate and the preserved value across parallel regions.

Data copying

- ***copying*:** similar to *firstprivate* for *private* variables, *threadprivate* variables are not initialized, unless using *copyin* to pass the value from the corresponding global variables. No*copyout* is needed because the value of a threadprivate variable is maintained throughout the execution of the whole program.
- ***copyprivate*:** used with *single* to support the copying of data values from private objects on one thread (the *single* thread) to the corresponding objects on other threads in the team.

Reduction

- ***reduction****(operator | intrinsic : list)*: the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable. This is very useful if a particular operation (specified in *operator* for this particular clause) on a datatype that runs iteratively so that its value at a particular iteration depends on its value at a prior iteration. Basically, the steps that lead up to the operational increment are parallelized, but the threads gather up and wait before updating the datatype, then increments the datatype in order so as to avoid racing condition. This would be required in parallelizing numerical integration of functions and differential equations, as a common example.

Others

- ***flush*:** The value of this variable is restored from the register to the memory for using this value outside of a parallel part
- ***master*:** Executed only by the master thread (the thread which forked off all the others during the execution of the OpenMP directive). No implicit barrier; other team members (threads) not required to reach.

User-level runtime routines

Used to modify/check the number of threads, detect if the execution context is in a parallel region, how many processors in current system, set/unset locks, timing functions, etc.

Environment variables

A method to alter the execution features of OpenMP applications. Used to control loop iterations scheduling, default number of threads, etc. For example *OMP_NUM_THREADS* is used to specify number of threads for an application.