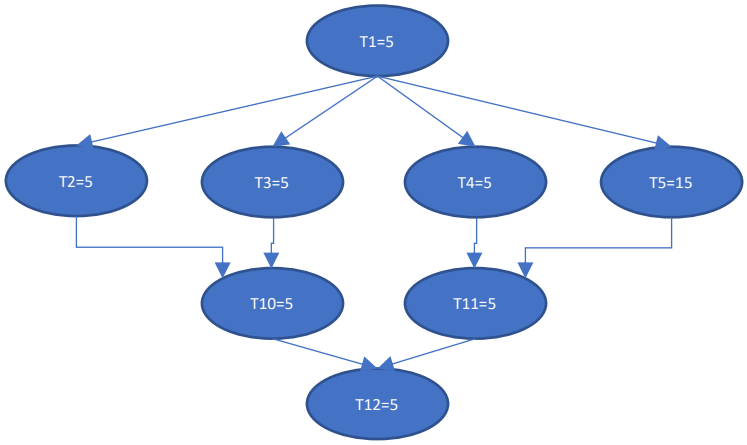
Conceptos básicos HPC

1) Dado el siguiente grafo de tareas contesta y justifica brevemente las siguientes preguntas



1) Calcula T(1), T(∞) sabiendo que el número que aparece junto a la tareas es el tiempo en segundos de la tarea. Indica la lista de tareas que forman parte del camino crítico

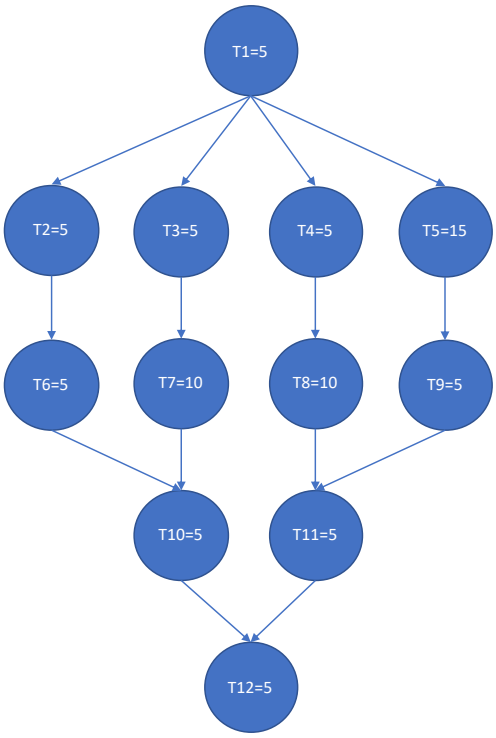
b) Calcula el *Paralelismo*

2) Contesta brevemente las siguientes preguntas

a) ¿Qué es el balanceo de carga?

b) Compara brevemente ventajas e inconvenientes de tener tareas muy pequeñas (fine grained) o grandes (coarse grained)

2) Dado el siguiente grafo de tareas contesta y justifica brevemente las siguientes preguntas



b) Calcula T(1) sabiendo que el número que aparece junto a la tareas es el tiempo en segundos de la tarea. Indica la lista de tareas que

c) Indica la lista de tareas que forman parte del camino crítico y justifica porque sono estas tareas y no otras.

d) Calcula T(∞)

e) Calcula el *Paralelismo*

3) Dado el siguiente código (incompleta y/o con errores), sabiendo que la inicialización de A, B, y el cálculo de C son totalmente paralelos, contesta razonadamente las siguientes preguntas:

```
1.  #define DIM_X 20
2.  #define DIM_Y 1000000
3.  double A[DIM_X][DIM_Y],B[DIM_X][DIM_Y],C[DIM_X][DIM_Y];
4.  void main(int argc,chat *argv[])
5.  {
6.     int i,j;
7.     #pragma omp parallel
8.     {
9.      #pragma omp single
10.    init_A(DIM_X,DIM_Y); /* Esta función inicializa A */
11.    /* primer loop */
12.     #pragma omp for
13.    for (i=0;i<DIM_X;i++){
14.       for (j=0;j<DIM_Y;j++)      update_A(A,i,j,DIM_X,DIM_Y);
15.    }
16.    #pragma omp single
17.    init_B(DIM_X,DIM_Y); /* Esta función inicializa B */
18.    /* segundo loop */
19.    #pragma omp for
20.    for (i=0;i<DIM_X;i++){
21.       for (j=0;j<DIM_Y;j++)      compute_C_AB(A,B,C,i,j,DIM_X,DIM_Y);
22.    }
23.    } /* Cerramos parallel */
24.    print_result(C,DIM_X,DIM_Y);
25. }
```

a) Calcula la fracción paralela que podemos deducir del análisis de este código basándote en las directivas que se han utilizado, y sabiendo que el coste de las funciones y bucles cuando utilizamos 1 thread es, respectivamente:
   - init_A → 1 segundo
   - Primer loop →197 segundos
   - init_B → 1 segundo
   - Segundo loop → 800 segundos
   - print_results → 1 segundos

b) ¿Cuál sería el máximo speedup que podríamos obtener en función de la fracción paralela que has calculado?

c) ¿Deberíamos modificar con algunas cláusula la visibilidad de las variables i,j en el bucle 1? ¿Y en el 2? Si propones alguna modificación indica cual y justifícala

d) Si ejecutamos este código en una máquina con 48 cores, ¿Qué problema nos encontraremos con la paralelización y el tamaño de datos actual? ¿Qué solución propondrías para aprovechar al máximo la cantidad de trabajo existente?

e) Dibuja el grafo de tareas que generaría este programa cuando lo ejecutamos con 10 threads. Asigna el coste de las tareas sabiendo que con 10 threads el programa escala perfectamente.

f) Calcula el speedup(10) usando como referencia los tiempos indicados en el apartado (a) y el grafo de tareas que has generado en el apartado anterior

g) Calcula la eficiencia (10) usando como referencia los datos calculados anteriormente

OpenMP

1) Indica las clausulas para especificar la visibilidad de las variables que hemos visto en clase para OpenMP y describe las principales características y diferencias entre ellas.

2) Dado el siguiente código, ¿cuántas instancias de la función do_work() ejecutará cada thread?

```
#pragma omp parallel for num_threads(48) collapse(2)
for (i=0;i<10;i++){
      for (j=0;j<48;j++){
      do_work(i);
      }
}
```

3) Dado el siguiente código, analízalo y contesta razonadamente las siguientes preguntas:

```
1.  #define DIM1 1000
2.  #define DIM2 1000
3.  double A[DIM1][DIM2];
4.  double f(double elem)
5.  {
6.  double x;
7.  /* calculo sobre elem */
8.  …
9.  return x;
10. }
11. void main(int argc,char *argv[])
12. {
13. double res=0;
14. int i,j,myid,totalth,init,end,num_iters,ch_per_th=10,ch;
15. init(A,DIM1,DIM2);
16. #pragma omp parallel  private(i,j,myid,totalth,init,end,num_iters,ch)
    firstprivate(ch_per_th) num_threads(2)
17. {
18. myid=omp_get_thread_num();
19. totalth=omp_get_num_threads();
20. num_iters=DIM1/(totalth*ch_per_th);
21. for (ch=0;ch<ch_per_th;ch++){
22.       init=myid*num_iters*ch_per_th+(ch*num_iters);
23.       end=init+num_iters;
24.       #pragma omp task firstprivate(init,end)
25.       {
26.       for (i=init;i<end;i++)
27.             for (j=0;j<DIM2;j++)
28.             #pragma omp atomic
29.             res=res+f(A[i][j]);
30.       } /* end task */
31. } /* for (ch.. */
32. } /* Parallel */
33. printf("res=%lf\n",res);
34. }
```

b) Describe brevemente que hace el código. Numero de threads que se crean,número de tasks que se crean así como el rango de iteraciones que ejecutará cada task (asigna un identificador a cada una de las tasks creadas)

c) Indica en que líneas de código encontramos alguna sincronización y si es explícita o implícita

d) Indica si podemos garantizar que tasks ejecutará cada thread. En caso afirmativo, indica que tasks ejecutará cada thread usando los identificadores del apartado (a)

e) El valor de la variable res, calculado en la línea 29, ¿Se conservará al acabar el cálculo y por lo tanto se imprimirá al final del programa?

MPI

1) ¿Qué entendemos por operación colectiva en MPI?

2) ¿Qué diferencia hay entre un MPI_Gather(..) y un MPI_Allgather(…)?

3) Nos dan el siguiente código MPI+OpenMP:

```
int main(int argc,char *argv[])
{
int ret,my_rank;
int var_compartida;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
#pragma omp parallel num_threads(4)
var_compartida=omp_get_thread_num();
printf("var_compartida=%d\n",var_compartida);
MPI_Finalize();
}
```

Y lo ejecutamos en un único nodo, con 4 procesos MPI.
¿Qué salida resultante de esta ejecución podemos esperar?

4) Dado el siguiente código (reducido para minimizar espacio), del cual sabemos que MATRIX_SIZE es múltiplo de 2, que lo vamos a ejecutar con 2 procesos MPI. El contenido exacto de read_data y compute_elem no es relevante.

```
double A[MATRIX_SIZE];
// Lee los datos y inicializa A
void read_data(double *A,int size)
{…
}
// Hace un calculo sobre el elemento y lo escribe en un fichero
void compute_elem(double *A)
{…
}
int main(int argc,char *argv[])
{
int ret,my_rank,my_size;
int i;
MPI_Request hand;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_rank(MPI_COMM_WORLD,&my_size);

if (my_rank==0){
             read_data(A,MATRIX_SIZE);
             MPI_Isend(&A[MATRIX_SIZE/2],MATRIX_SIZE/2,MPI_DOU
             BLE,1,MPI_ANY_TAG,MPI_COMM_WORLD,&hand);
}else{
             MPI_Irecv(A,MATRIX_SIZE/2,MPI_DOUBLE,0,MPI_ANY_TA
             G,MPI_COMM_WORLD,&hand);
}
// Bucle de calculo
for (i=0;i<MATRIX_SIZE/2;i++){
             compute_elem(&A[i]);
}
MPI_Finalize();
}
```

Ejecutamos el programa y vemos que tenemos un problema y los datos resultantes no son los correctos.

a) ¿A qué crees que es debido?

b) ¿Cómo primera solución se nos ocurre poner justo después del MPI_Isend(…) un MPI_Wait(&hand,&status), ¿solucionaría esto el problema? Justifícalo y, en caso negativo, indica cómo habría que solucionarlo?

c) ¿Otra alternativa que se nos ocurre poner justo antes del comentario (`Bucle de calculo`) un MPI_Barrier(MPI_COMM_WORLD), ¿solucionaría esto el problema? Justifícalo y, en caso negativo, indica cómo habría que solucionarlo?

d) Queremos hacer una versión nueva del programa anterior con una aproximación diferente. En este caso será más genérico: soportará N procesos MPI y la matriz estará en un fichero, del cual cada proceso leerá directamente una parte. Nos piden completar la función `compute_dimensions`(..) que calcula la parte del fichero que corresponde a cada proceso.

```
double *A;
int file_size(char *file_name) {…}
void compute_dimensions(int my_size,int my_rank,int fsize,int
*init,int *end,int *chunk){…}
void read_data(double **A,char *file_name,int init,int end){…}
void compute_elem(double *A) {…}
int main(int argc,char *argv[])
{
int ret,my_rank,my_size,chunk,fsize,init,end;
int i;
MPI_Request hand;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_rank(MPI_COMM_WORLD,&my_size);

// file_size calcula el tamaño de la matriz
fsize=file_size(argv[1]);
// Calcula parte de fichero que corresponde al proceso
compute_dimensions(my_size,my_rank,fsize,&init,&end,&chunk);
// Reserva espacio para A y lee datos
read_data(&A,argv[1],init,end);
// Bucle de calculo
for (i=0;i<chunk;i++){
            compute_elem(&A[i]);
}
MPI_Finalize();
}
```

```
void compute_dimensions(int my_size,int my_rank,int fsize,int
*init,int *end,int *chunk)
{




}
```

5) Contesta brevemente las siguientes preguntas
   a) ¿Qué es un comunicador en MPI? ¿Para qué se utilizan? ¿Puede un mismo proceso MPI estar en dos o más comunicadores a la vez?

   b) ¿Qué diferencia hay entre comunicaciones síncronas y asíncronas

   c) ¿Qué entendemos por operación colectiva en MPI?

   d) ¿Qué diferencia hay entre un MPI_Broadcast(..) y un MPI_Scatter(..)?

6) Nos dan el código MPI de la siguiente figura, que ejecutamos en un único nodo, con 48 procesos MPI. ¿Qué salida resultante de esta ejecución podemos esperar? ¿Imprimirá cada proceso su propio PID o imprimirán todos el PID del último que haya hecho la modificación?

```
int main(int argc,char *argv[])
{
      int ret,my_rank;
      int var_compartida;
      var_compartida=getpid();
      MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
      var_compartida=getpid();
      MPI_Barrier(MPI_COMM_WORLD);
      printf("var_compartida=%d\n",var_compartida);
      MPI_Finalize();
}
```