Cálculos en paralelo con FreeFem++

J. Rafael Rodríguez Galván

4 al 8 de julio de 2011

Outline

1. Cálculos en paralelo y paralelización
2. FreeFem++ y MPI
3. Uso de FreeFem++ en el cluster de la Universidad de Cádiz

Tipos de sistemas paralelos

- Memoria compartida (UMA, *uniform access to memory* o SMP *symetric multiprocessing*).

  Varios procesadores acceden a un mismo espacio de memoria, a través de una interfaz de red

  - *Ejemplo*: ordenadores personales con varios procesadores o núcleos
- Memoria no compartida (NUMA, *non uniform access to memory*).

  Existen varios procesadores, cada uno de ellos con su propia memoria. Los procesadores se conectan entre sí a través de una interfaz de red.

  - *Ejemplo*: cluster de la UCA
  - *Ventajas*: menor coste, mayor flexibilidad para dimensionar la máquina
  - *Inconvenientes*: mayor latencia en la sincronización entre procesadores, necesidad de usar bibliotecas de paso de mensajes

Algoritmos y sistemas paralelos

- **Idea**: realizar una serie de cálculos de forma independiente, en varios procesadores
- La forma de ponerla en práctica, depende del tipo de sistema paralelo en el que vayamos a realizar los cálculos:
  - Dividir el proceso en distintos **hilos de ejecución** (*threads*) (o subprocesos en un mismo espacio de memoria)
    - Para ello, debemos usar una biblioteca adecuada
    - Sólo en sistemas de memoria compartida
  - Usar una biblioteca de **paso de mensajes** y ejecutar procesos sincronizados (en espacios de memoria diferentes)
    - Para ello, debemos usar una biblioteca adecuada
    - En sistemas de memoria compartida o distribuida
  - Realizar cálculos paralelos **sin sincronización**
    - Por ejemplo, ejecutar un mismo programa con distintos datos de entrada
    - En sistemas de memoria compartida o distribuida

Bibliotecas de paso de mensaje: *MPI*

- Es la técnica que ofrece mayor flexibilidad: funciona en todo tipo de máquinas, con memoria compartida o no
- **MPI** es el estándar para el paso de mensajes de uso habitual
- El estándar *MPI* define 125 funciones, pero sólo 6 son esenciales: `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Recv`, `MPI_Finalize`
- Existen bastantes implementaciones: *lam*, *mpich*, *openmpi*
- Y bastantes lenguajes capaces de usar *MPI*: C/C++, Fortran, Python,... y **FreeFem++**

MPI en la práctica

- Es necesario tener instaladas la biblioteca *MPI*
- Es necesario programar usando las funciones de *MPI*
- Es necesario ejecutar el programa utilizando `mpirun`
  - En **lenguaje C**:
    - Compilación:
      `mpicc -l mpi -o mpiprograma miprograma.c`
    - Ejecución con 4 procesos:
      `mpirun -np 4 miprograma`
  - En **lenguaje FreeFem++**
    - Existe un intérprete compilado con *MPI*: `FreeFem++-mpi`
    - Ejecución con 4 procesos:
      `mpirun -np 4 FreeFem++-mpi miprograma.edp`
  - El programa “hola mundo paralelo” en FreeFem++

```
1  cout << "Hola, soy el proceso " << mpirank
2       << " de " << mpisize << endl;
```

Y ¿tengo que aprender *MPI* para usar FreeFem++ en un cluster?

Respuesta corta: **No**.
Respuesta larga: existen varias posibilidades, en las dos primeras no es necesario programar, expresamente, con *MPI*:

1. Utilizar el cluster para cálculos paralelos **sin sincronización**
   - Por ejemplo, ejecutar un mismo programa con distintos datos de entrada (distintas mallas, distintos parámetros...)
2. Utilizar una biblioteca paralela (con la opción `solve`) para resolver los **sistemas de ecuaciones de forma distribuida**
3. Aprender el subconjunto de *MPI* que implementa `FreeFem++-mpi` y programar usando estas funciones
   - Única solución que permite diseñar algoritmos paralelos *a medida*
   - Solución de mayor potencia, pero requiere programar los algoritmos específicamente
   - Un ejemplo...

El algoritmo de Schwartz con *MPI*

Disponible en FreeFem++: `ejemplos++-mpi/schwartz.edp`).

```
if ( mpisize != 2 ) { // ¡Exactamente 2 procesadores!
  cout << " sorry number of processeur !=2 " << endl;
  exit(1); }
verbosity=3;
real pi=4*atan(1);
// Construcción de dos mallas con solapamiento:
int inside = 2; int outside = 1;
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t ;y=1;label=outside;};
border d(t=1,0){x = 1-t; y = t;label=inside;};
border e(t=0, pi/2){x=cos(t);y=sin(t);label=inside;};
border e1(t=pi/2, 2*pi){ x= cos(t); y = sin(t);label=
   outside;};
int n=50;
mesh th,TH;
```

El algoritmo de Schwartz con *MPI* (II)

```
// Cada procesador construye una malla...
if (mpirank == 0)
 {
 th = buildmesh( a(5*n) + b(5*n) + c(10*n) + d(5*n));
 cout << " end th  " << endl;
}
else
 {
 TH = buildmesh ( e(5*n) + e1(25*n) );
 cout << " end TH  " << endl;
 }
// ... y se la pasa al resto del mundo (o sea, al otro)
broadcast(processor(0),th);
broadcast(processor(1),TH);
```

El algoritmo de Schwartz con *MPI* (III)

```
// Definición de los espacios de elementos finitos...
fespace vh(th,P1);
fespace VH(TH,P1);
vh u=0,v; VH U,V;
int i=0;
// ... y de los problemas variacionales, en cada malla
problem PB(U,V,init=i,solver=Cholesky) =
    int2d(TH)( dx(U)*dx(V)+dy(U)*dy(V) )
  + int2d(TH)( -V) + on(outside, U=0) + on(inside, U=u);
problem pb(u,v,init=i,solver=Cholesky) =
    int2d(th)( dx(u)*dx(v)+dy(u)*dy(v) )
  + int2d(th)( -v) + on(outside, u=0) + on(inside, u=U);

// Obsérvese cómo una condición de contorno (Dirichlet)
// acopla la solución con la obtenida en la otra malla.
```

El algoritmo de Schwartz con *MPI* (IV)

```
// Método iterativo: cada procesador resuelve y envía la
// solución al otro (que la usará como dato de contorno)
for ( i=0 ;i< 10; i++)
{
  cout << mpirank << " looP " << i << endl;
  if (mpirank == 0)  {
   PB;
   processor(1) << U[];
   processor(1) >> u[];
  }
  else    {
   pb;
   processor(0) >> U[];
   processor(0) << u[];
  }
};  // Fin, el 1er proceso se encarga de grabar:
if (mpirank==0) plot(U, u, ps="Uu.eps");
```

Bibliotecas paralelas para sistemas lineales

- Bibliotecas externas a *FreeFem++*, especializadas en la resolución de sistemas lineales usando algoritmos paralelos
  - `MUMPS_FreeFem`
  - `real_SuperLU_DIST_FreeFem`
  - `complex_SuperLU_DIST_FreeFem`
  - `real_pastix_FreeFem`
  - `complex_pastix_FreeFem`
  - `hips_FreeFem`
  - `hypre_FreeFem`
  - `parms_FreeFem`
- Cada una tiene sus características y parámetros de configuración
- Ejemplo (ver FreeFem++: `ejemplos++-mpi/Stokes-*.edp`)

```
1  load "real_SuperLU_DIST_FreeFem"
2  string ssparams="nprow=1, npcol="+mpisize;
3  // ...(definición de M y b)...
4  set(M,solver=sparsesolver, sparams=ssparams);
5  u = M^-1 * b;
```

El *cluster* de la Universidad de Cádiz

http://supercomputacion.uca.es

- 80 máquinas, cada una con 4 núcleos (320 en total)
- 640 GB de memoria principal
- Sistema operativo Linux
  - No es capricho, el 91 % del 500 las máquinas más potentes del mundo [http://www.top500.org] usan este sistema operativo
- Manual de uso: http://supercomputacion.uca.es/curso_usuarios_2007.pdf
- Sistema de colas *condor* (sobre *MPI*)
  - Permite distribuir un proceso en paralelo entre aquellos nodos en los que sea más conveniente y en el momento adecuado
  - *Universo vanilla*: permite la ejecución de proceso en varios nodos (procesadores) sin intercomunicación.
  - *Universo parallel*: permite la ejecución de un proceso en varios nodos con intercomunicación (mediante *MPI*)

Ejemplo de uso de FreeFem++ en el *cluster*

- Usaremos el *universo vanilla* para ejecutar un programa FreeFem++
  - En 4 procesadores diferentes
  - Con distintos datos
- El programa, `hola.edp`, lee un fichero de datos (identificado por el 3er argumento de entrada, `ARGV[3]`) y muestra su contenido

```
1  cout << "Hola" << endl;
2  // ARGV[0]->"condor_exec.exe", ARGV[1]->"-nw"
3  // ARGV[2]->"hola.edp"
4  string process = ARGV[3];//ARGV[3]->N° proceso actual
5  string inputFile = "infile." + process;
6  cout << "Input file: " << inputFile << endl;
7  string data; // To read input file
8  { ifstream ifs(inputFile); getline(ifs, data); }
9  // Write data to sandard output
10 cout << "Mis datos:" << data << endl;
11 cout << "Adios" << endl;
```

Fichero para ejecutar el programa (*universo vanilla*)

- Para ejecutar[1] el programa FreeFem++ en el sistema *condor* usaremos el siguiente fichero (al que llamaremos `vanilla.sub`)

```
1  universe = vanilla
2  initialdir = .
3  executable = /home/software/bin/FreeFem++
4  arguments = -nw hola.edp $(Process)
5  output = outfile.$(Process)
6  error  = errfile.$(Process)
7  log  = log.txt
8  queue 4
```

- Línea 4: argumentos que serán pasados al programa ejecutable (*FreeFem++*), desde donde se leen con `ARGV[i]`
- La variable `$(Process)` tomará un valor que identifica a el nodo de ejecución (valores $0, 1, ..., N-1$, con $N = 4$)
- Línea 5: fichero donde se grabará la salida del programa
- Línea 8: número de procesos paralelos que lanzaremos

Ejecución del programa

- Supongamos que, en el mismo directorio que `vanilla.sub`, tenemos 4 ficheros llamados `infile.0`, ..., `infile.3` (según hemos especificado en la línea 5 de `hola.edp`)
- Ejecutamos el programa usando el programa `condor_submit`.

```
1  > condor_submit vanilla.sub
2  Submitting job(s)..
3  Logging submit event(s)..
4  4 job(s) submitted to cluster 27719.
```

- Con `condor_q` vemos los procesos en cola (ver documentación).
- Al finalizar, recibiremos un correo interno al cluster (podemos leerlo con `mutt`). En el fichero `.sub` podemos especificar otra dirección de correo personal (variable `notify_user`).
- La salida de cada proceso se guardó en un fichero `outfile.*`

Comentario final: exportar y visualizar ficheros VTK

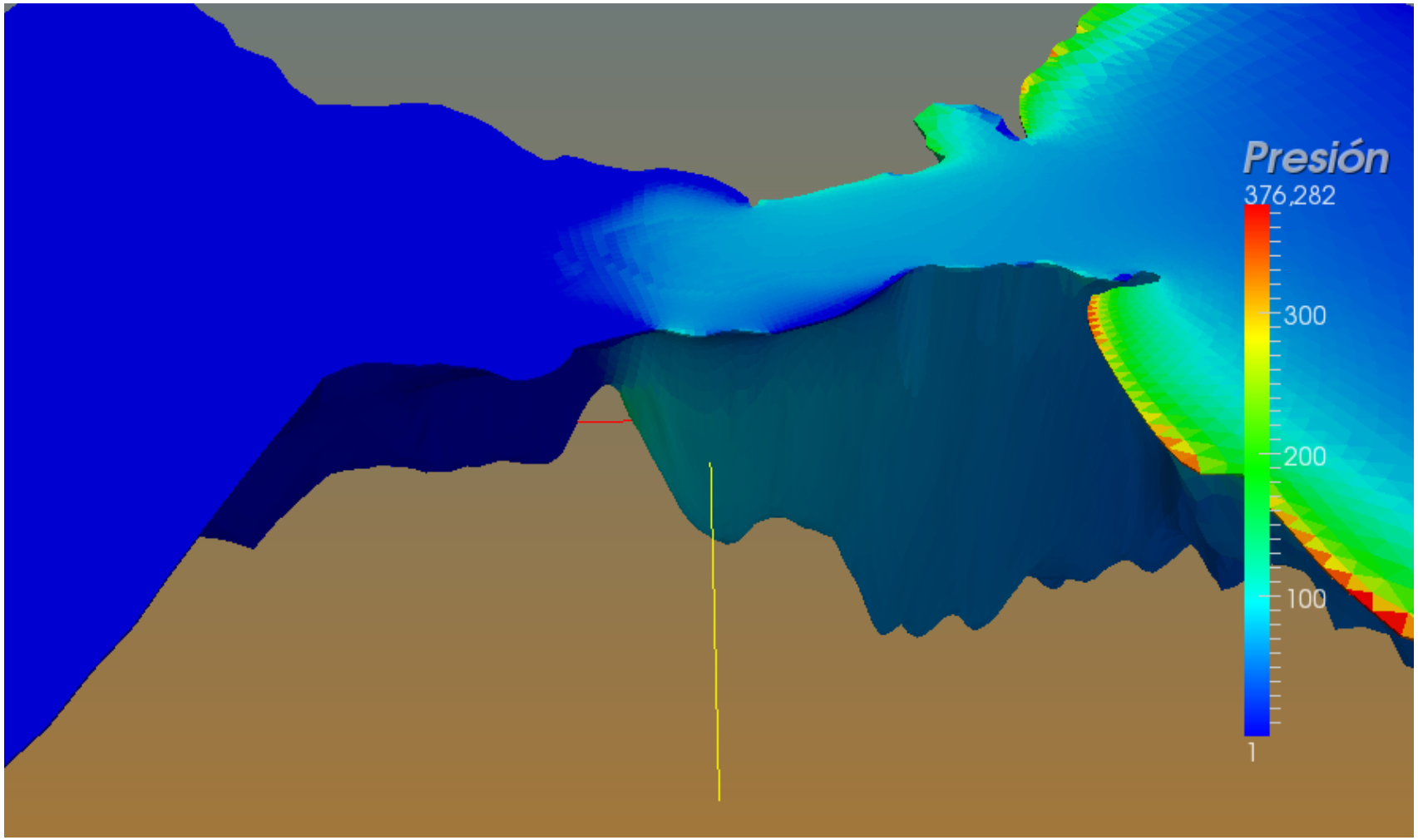
- El ejemplo `hola.edp` es muy simple: lo habitual en *FreeFem++* es que nuestro programa resuelva un problema variacional. ¿Y luego? El cluster no es un entorno para visualizar los resultados.
- Una posibilidad: guardar (con `ofstream`) la solución a un fichero para post-procesarla y/o visualizarla fuera del cluster.
- Otra posibilidad: **guardar la solución en formato vtk**: un tipo de ficheros especializado en la visualización de imágenes 3D.

```
1  load "iovtk"
2  // Resolvemos, por ejemplo, el pb de Stokes en Th,
3  // obteniendo velocidad (u1, u2, u3) y presión (p),
4  // y grabamos el resultado en formato vtk:
5  savevtk("stokes.vtk", Th, [u1, u2, u3], p);
```

- Ahora, copiamos a nuestro ordenador el fichero `stokes.vtk` y **podemos visualizarlo en 3D con Paraview** o **Mayavi2**[2].

[2] Excelentes programas (libres) para la visualización de ficheros *vtk* 2D/3D

Un ejemplo 3D con VTK (generado con *Paraview*)



Presión en el estrecho de Gibraltar producida por el viento superficial (condición de contorno Neumann para el problema de Stokes).