Programación en Paralelo con MPI en Clusters Linux

Francisco Javier Rodríguez Arias

13 de marzo de 2006

Problema y Motivación

En física se requiere hacer muchos cálculos. Para eso se hacen programas de cálculo, normalmente los hacemos en Fortran ó C/C++.

Específicamente en una aplicación que tenía, requería un tiempo de ejecución extremádamente grande. Esto motivó a que buscara una solución a ese problema. La solución fue paralelizarlo.

¿Qué es programación en paralelo?

Programación en paralelo es una técnica de programación, que permite ejecutar una misma tarea en distintos procesadores.

Se pueden clasificar distintos modelos de programación en paralelo:

- Paralelizar Data: Conocido como Single Instruction/Multiple Data (SIMD). Un mismo conjunto de instrucciones se aplica simultáneamente sobre distintos datos.
- Paralelizar Tareas: Conocido como Multiple Instruction/-Multiple Data (MIMD). Distintas instrucciones se aplican sobre datos distintos.

Otros modelos

Otro modelo es el de SPMD (Single Program/Multiple Data) que en realidad puede reducirse al caso MIMD.

Implementaciones

El MPI está diseñado para MIMD. Mientras que el HPF u OpenMP están diseñado más para SIMD.

Ejemplos de SIMD

En SIMD cada proceso se encarga de una parte de la data, aplicándose sobre cada parte las mismas instrucciones.

- Operaciones aritméticas vectoriales/matriciales.

¿Qué es un Cluster?

Un Cluster de computadoras es un conjunto de computadoras conectadas entre sí, que trabajan una tarea relacionada, de tal forma que parecen *una gran computadora*. Cada procesador de cada computadora se puede considerar un nodo del Cluster.

Cuando se habla de programación en paralelo, se entiende que los distintos procesos que se quieren *ejecutar* en paralelo se distribuirán a través de los distintos nodos de un Cluster. Entonces podemos tener computadoras con multiprocesador, que tendrían varios nodos del Cluster.

Utilidad

La mayor utilidad de la paralelización es la ganancia en velocidad de la ejecución.

Hay que tener en cuenta que no todo problema/algoritmo puede ser paralelizable. Y si lo es, no necesariamente hay una ganancia en velocidad, puesto que debemos considerar el tiempo de comunicación entre los procesos.

Un modelo simple de la velocidad de un programa en paralelo es:

$$T = \frac{T_p}{p} + T_s + T_c$$

- T: Tiempo total.
- T_p: Tiempo de la parte paralelizable.
- p: Cantidad de procesadores.
- T_s: Tiempo de la parte serial (no paralelizable).
- T_c: Tiempo de comunicación. Es 0 cuando $p = 1$.

¿Qué es MPI?

MPI = Message Passing Interface

- Es una librería independiente del lenguaje de programación. Existen implementaciones para Fortran, C y C++.
- Facilita la comunicación entre procesos distintos.
- Funciona haciendo un *fork* del programa. Pudiéndose distribuir los procesos en distintos procesadores, ubicados en una o varias máquinas.

¿Qué ventajas tiene el MPI?

El MPI permite distribuir sus nodos tanto en distintas computadoras como en distintos procesadores en una misma computadora. Permite también que se asignen varios procesos a un mismo procesador, así pues si se tiene un Cluster heterogéneo de computadoras, se puede, por ejemplo, asignar más procesos (es decir, más tareas) a los procesadores más rápidos.

Implementación usada

La implementación del MPI usada fue LAM/MPI, que es una implementación del estándar MPI (implementado el MPI-1.2 completo, y muchas cosas del MPI-2).

Esta implementación está hecha sólo para GNU/Linux, lo que hace una necesidad el usar ese sistema operativo en todas las computadoras que conformen parte del Cluster.

Instrucciones básicas del MPI

Las instrucciones básicas del MPI cubren 3 aspectos:

- Inicialización y terminación del MPI.
- Identificación de quién soy y cuántos somos.
- Envío y recepción de mensajes entre los procesos.

Inicialización y terminación del MPI

Todo programa MPI debe comenzar con la instrucción:

```
MPI_Init(&argc, &argv);
```

Y debe terminar con:

```
MPI_Finalize();
```

Identificación de quién soy y cuántos somos

Con las siguientes instrucciones se puede obtener la información básica de los nodos, es decir, cuántos son, y cuál soy.

```
int myrank; // Numero de proceso.
int nprocs; // Numero total de procesos.

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

Envío y recepción de mensajes entre los procesos

Las instrucciones que soportan el esqueleto de todo programa MPI son:

Enviar mensaje:

```
MPI_Send(buffer, contador, tipodato, destino,
         tag, MPI_COMM_WORLD);
```

Recibir mensaje:

```
MPI_Recv(buffer, maxcontador, tipodato,
         origen, tag, MPI_COMM_WORLD,
         &status);
```

Instrucciones colectivas

El MPI provee de instrucciones colectivas, que se aplican a todos los procesos al mismo tiempo. Las funciones colectivas básicas son las necesarias para enviar información desde un proceso al mismo tiempo a todos los demás, y recibir, aplicando una operación, la información de todos los procesos en uno solo.

Éstas son:

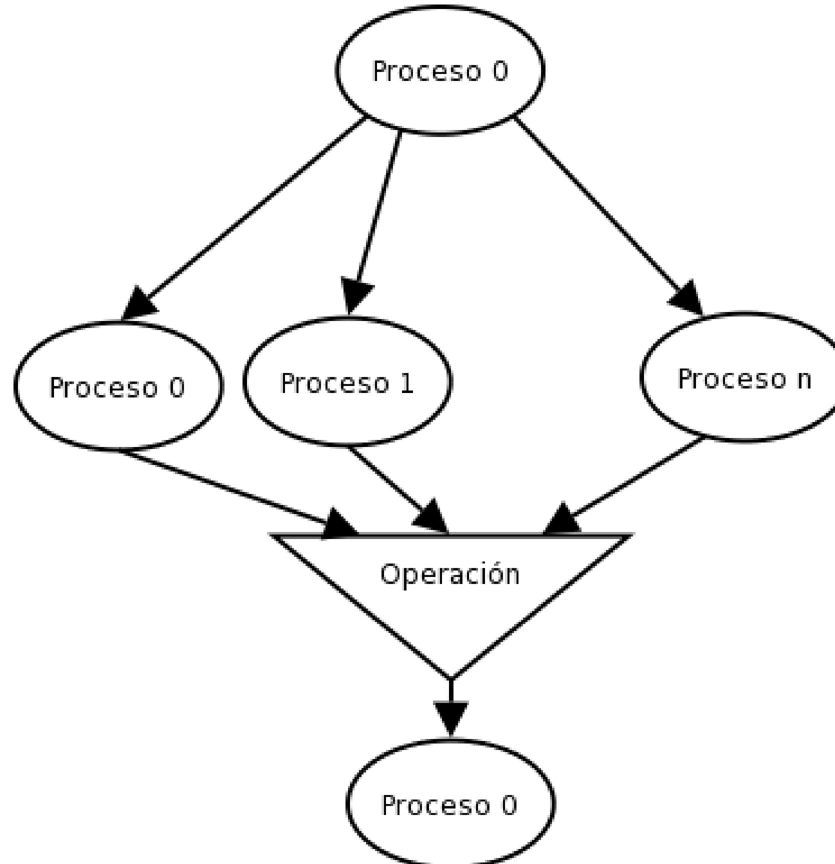
```
MPI_Bcast(buffer, contador, tipodato,
          root, MPI_COMM_WORLD);
MPI_Reduce(sendbuf, recvbuf, contador,
           tipodato, operacion, root,
           MPI_COMM_WORLD);
```

Ejemplo básico: cálculo de PI

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)
  {
    if (myid == 0)
    {
      printf("Introduzca␣la␣cantidad␣de␣intervalos:␣(0␣sale)␣");
      scanf(" %d",&n);
```

```
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi es aproximadamente %.16f, Error %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

Esquema del ejemplo



Desventajas de ese esquema

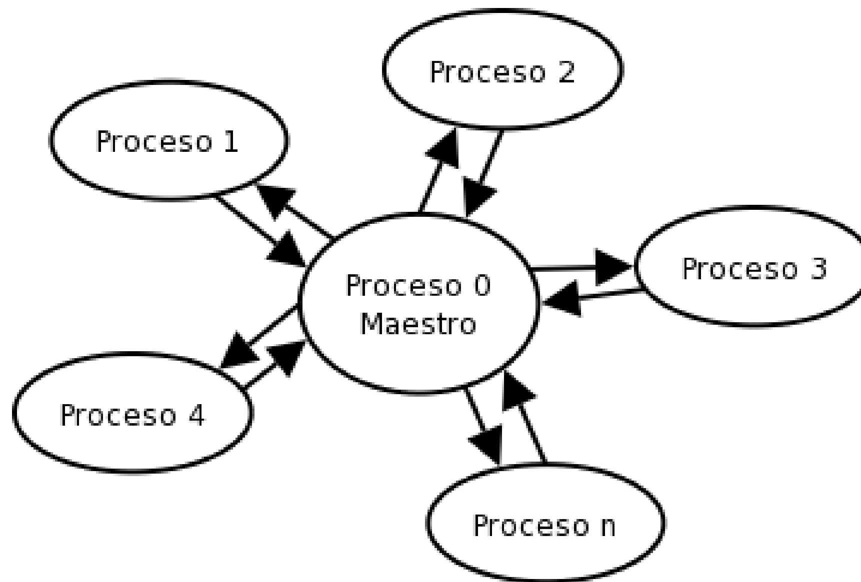
El esquema utilizado en el ejemplo, es considerado un esquema de paralelización eficiente, pues divide en forma simple y equitativa la tarea.

La desventaja que puede tener es que el tiempo de la parte paralelizada, será lo que se demore el proceso más lento. Esto es especialmente fastidioso si el cluster es muy heterogéneo y cuenta con procesadores de distintas velocidades.

Dependerá el problema a resolver el esquema de paralelización a usar.

Maestro/Esclavo

En el esquema maestro/esclavo, se tiene un proceso principal (maestro) que se encarga de asignarle tareas a los demás procesos (esclavos).



Aplicación Real: Especificaciones del problema

Para un proyecto de un curso, que luego formó parte de un proyecto de la DAI, necesitaba hacer unos cálculos ligeramente complejos. Esa parte se titulaba *"Aspectos Fenomenológicos de la Física de Neutrinos Masivos"*.

Necesitaba calcular un espectro (una especie de histograma), que era resultado de la convolución de una integral (es decir, una doble integral), en que para cada punto de la curva sobre la que se integraba, se necesitaba resolver una sumatoria doble, y en la cual, en cada elemento se necesitaba la solución de una ecuación diferencial matricial (3x3 de números complejos, que se resolvía usando Runge Kutta de orden 4). (Tiempo aproximado por cálculo $\approx 1 \sim 2$ min).

Además, ese espectro se necesitaba obtener para ciertos parámetros dados (6 parámetros), que se variaban en una malla de puntos (de 6 dimensiones). Por simplicidad, se fijaban *cortes* de estos parámetros, y se dejaban dos parámetros, que serían los que se variarían. Entonces para cada par de valores de estos parámetros se realizaba el cálculo mencionado.

Para cada corte, se necesitaban en un primer cálculo, unos 10000 puntos para obtener resultados razonables.

Considerando 1 min por punto, se necesitarían 166 horas para obtener el resultado final de un corte.

Aplicación Real: Herramientas usadas

Para resolver ese problema, se trabajó en un entorno Red Hat.

- Se usaron los siguientes compiladores: g++, gcc, g77.
- La implementación de MPI usada fue LAM/MPI.
- Para la generación de los gráficos se usó: GNUPlot, PAW y Root.

El conjunto de programas desarrollados consta de más de 13000 líneas de código.

Aplicación Real: Atacando el problema

En este problema, muchas fases del cálculo podían ser paralelizables. Las integrales, las sumatorias, las ecuación diferencial. Pero para simplificarlo, reduje la paralelización, a nivel del cálculo de un punto de la malla. Atacaría el problema usando un esquema maestro/esclavo, en el cual la tarea nuclear sería el cálculo de un punto de la malla.

Bajo este esquema, se tiene un proceso maestro, que es el que tiene en su espacio de memoria, toda la malla, y manda a cada proceso esclavo la tarea de calcular el espectro correspondiente para un punto de esa malla.

Detallando esquema

Para detallar mejor el esquema, aquí está el pseudocódigo de la parte paralelizada:

- Inicializar MPI
- Identificarse como proceso
- Si soy maestro (proceso 0), generar lista (malla) y hacer la Tarea Maestro.
- Si no soy maestro, soy esclavo, entonces hacer la Tarea Esclavo.

Tarea Maestro

- Inicializar seguir como la cantidad de procesos más la cantidad de puntos.
- Inicializar n $= 0$
- Mientras seguir > 0
  - Recibir cualquier mensaje de cualquier proceso.
  - Si identificador de mensaje es *Estoy Libre* entonces
    - Si n $<$ cantidad de tareas (es decir, falta tareas por hacer) entonces
      - Enviar mensaje al proceso del que se recibió (desde ahora *source*) informando que se le mandará una tarea.
      - Enviar a *source* el número de tarea.
      - Enviar a *source* la información de la tarea.
      - Incrementar n.

  - Si no hay más tareas
    - Enviar mensaje a *source* informando que no hay más tareas, que ya pare su ejecución.
    - Decrementar seguir.
- Si identificador de mensaje es *Mando Solución de Tarea*, entonces
  - Recibir número de tarea de *source*.
  - Recibir el objeto con la data de la tarea.
  - Decrementar seguir.

Tarea Esclavo

- Mientras seguir
  - Enviar mensaje a maestro con identificador *Estoy Libre*
  - Recibir mensaje de maestro, que le dirá si quedan o no tareas, es decir si va a seguir o no
  - Si seguir, entonces
    - Recibir de maestro el número de tarea.
    - Recibir de maestro la data de la tarea.
    - Calcular el resultado de la tarea pedida.
    - Enviar mensaje a maestro con identificador *Mando Solución de Tarea*
    - Mandar número de tarea
    - Mandar data de la tarea, incluyendo resultados.

Código

Aquí están las partes importantes del código. Este primer método es ejecutado en todos los procesos:

```
int kamland::generalistaTODOa(const setdata estadoinicial,
                              const setdata estadofinal,
                              const setdata cantidades,
                              const setdata tipocambio,
                              int baseusada)
{
  int indice, i, j, k, ini, nuevoselementos;

/* Variables sobre la malla */

  kamlandtest *anterior;

  elementosultimatarea = 0;
```

```
  int quiensoy, cuantossomos;
  tiempoultimatarea = MPI_Wtime();
  MPI_Comm_size ( MPI_COMM_WORLD, &cuantossomos );
  MPI_Comm_rank ( MPI_COMM_WORLD, &quiensoy );
  if ((cuantossomos > 1) && (quiensoy != 0))
    return esclavogeneralistaTODO(quiensoy);

/*
   aqui esta el codigo para generar la malla
   obviado por ser varias paginas que no vienen
   al caso.
*/

  if (cuantossomos > 1)
    return maestrogeneralistaTODO(ini, ini + nuevoselementos,
                                  cuantossomos);

  for(i = 0; i < nuevoselementos; i++)
  {
    indice = ini + i;
```

```
    generaelemento(indice); // Calculo de LA TAREA
  }

  tiempoultimatarea = MPI_Wtime() - tiempoultimatarea;
  return 0;
}
```

Código: Tarea Maestro

```
int kamland::maestrogeneralistaTODO(int inicial, int final,
                                    int cuantos)
{
  int seguir, n, siseguir, noseguir;
  MPI_Status status;
  siseguir = 1;
  noseguir = 0;
  seguir = cuantos - 1 + final - inicial;
  n = inicial;
  while(seguir)
  {
    MPI_Recv(0, 0, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
    if (status.MPI_TAG == KTAGLIBRE)
    {
      if (n < final)
      {
```

```
      MPI_Send(&siseguir, 1, MPI_INT, status.MPI_SOURCE,
               KTAGLIBRE, MPI_COMM_WORLD);
      MPI_Send(&n, 1, MPI_INT, status.MPI_SOURCE, KTAGINFO,
               MPI_COMM_WORLD);
      lista[n].enviarme(status.MPI_SOURCE, KTAGINFO, 2);
      n++;
    }
    else
    {
      MPI_Send(&noseguir, 1, MPI_INT, status.MPI_SOURCE,
               KTAGLIBRE, MPI_COMM_WORLD);
      seguir--;
    }
  }
  else if (status.MPI_TAG == KTAGMANDO)
  {
    int i;
    MPI_Recv(&i, 1, MPI_INT, status.MPI_SOURCE, KTAGINFO,
             MPI_COMM_WORLD, &status);
    lista[i].recibirme(status.MPI_SOURCE, KTAGINFO, 1);
```

```
      lista[i].kcurva.datax = testbase.kcurva.datax;
      lista[i].kcurva.externodatax = 1;
      seguir--;
    }

  }
  tiempoultimatarea = MPI_Wtime() - tiempoultimatarea;
  return 0;
}
```

Código: Tarea Esclavo

```
int kamland::esclavogeneralistaTODO(int numeroesclavo)
{
  int seguir, n;
  MPI_Status status;
  lista = new kamlandtest[1];
  while (seguir)
  {
    MPI_Send(0, 0, MPI_INT, 0, KTAGLIBRE, MPI_COMM_WORLD);
    MPI_Recv(&seguir, 1, MPI_INT, 0, KTAGLIBRE, MPI_COMM_WORLD,
             &status);
    if (seguir)
    {
      MPI_Recv(&n, 1, MPI_INT, 0, KTAGINFO, MPI_COMM_WORLD, &status);
      lista[0].recibirme(0, KTAGINFO, 2);
      generaelemento(0); // Calculo de LA TAREA
      elementosultimatarea++;
      MPI_Send(0, 0, MPI_INT, 0, KTAGMANDO, MPI_COMM_WORLD);
```

```
      MPI_Send(&n, 1, MPI_INT, 0, KTAGINFO, MPI_COMM_WORLD);
      lista[0].enviarme(0, KTAGINFO, 1);
    }
  }
  tiempoultimatarea = MPI_Wtime() - tiempoultimatarea;
  return numeroesclavo;
}
```

Referencias

- http://www.lam-mpi.com
- http://www.cecalc.ula.ve/HPCLC/sitio_html/i_program.html