Paralelismo

- El objetivo es:
  - Tardar menos tiempo
  - Poder resolver problemas más grandes
- Soluciones:
  - Optimizar código (a mano/ con compiladores)
  - Optimizar algoritmos
  - Usar paralelismo

Optimizando

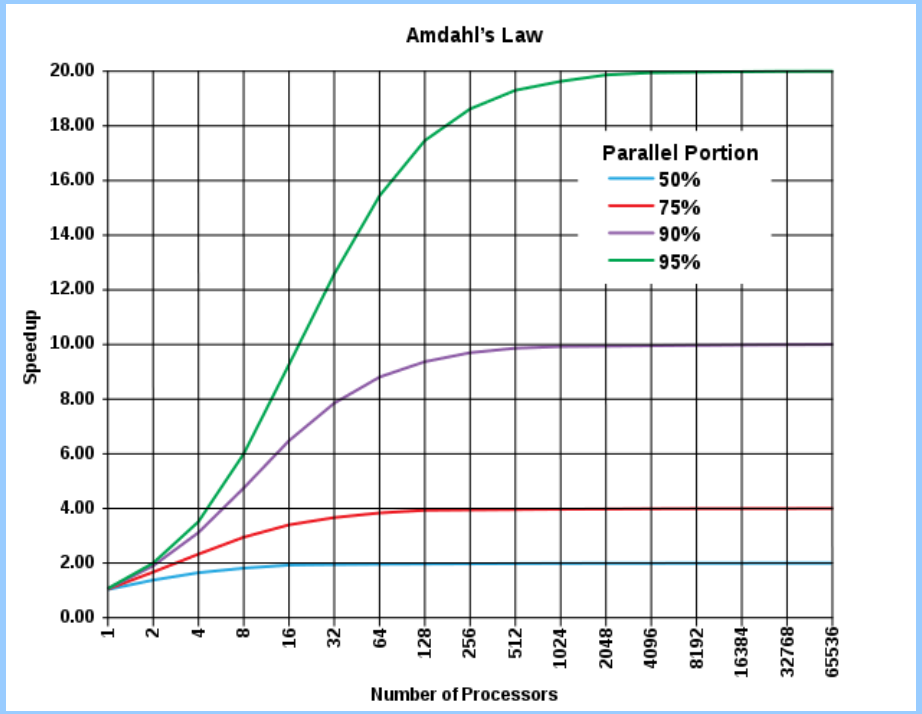
Las opciones son las mismas en C/C++ y en fortran :

- Para optimización “normal” usar el argumento -fast
  - Incluye: -O3 -ipo -no-prec-div -mdynamic-no-pic
  - -no-prec-div : hace las divisiones x/y como x*(1/y)
    - gana un poco de velocidad pero pierde un poco de precisión
    - para que las haga de forma normal poner detrás: -prec-div
- Gran variedad de optimizaciones:
  - Propagación de constantes
  - Eliminación de código que no se ejecuta
  - Asignación global de registros
  - Planificación de instrucciones y control especulativo de saltos global.
  - desenrollado de bucles global
  - eliminación de la redundancia parcial
  - renombrado de variables
  - optimización “peephole” (de pequeños trozos de código)

Ley de Amdahl

- P es la parte del programa que se puede hacer paralela
- N es el número de cores a usar

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}.$$



Paralelización automática

Se le puede pedir que realice una paralelización automática con -parallel, hay muchos casos en los que no va bien, suele funcionar mejor en fortran, gracias a no usar punteros.

Código original en fortran:

```
do i=1,100
 a(i) = a(i) + b(i) * c(i)
 enddo
```

Código paralelizado que se ejecuta tras compilar con -parallel :

Thread 1 :

```
do i=1,50
 a(i) = a(i) + b(i) * c(i)
 enddo
```

Thread 2 :

```
do i=51,100
 a(i) = a(i) + b(i) * c(i)
 enddo
```

Profiling para optimizar

Se basa en ejecutar el programa a optimizar para saber como optimizarlo mejor, que trozos de código se llaman mas, que decisiones se toman con mas frecuencia, etc.

Se divide en tres fases:

1: Compilación incluyendo la instrumentación del código y linkado usando: -prof_gen

2: Ejecución del programa, que generará ficheros con la información de la instrumentación.

3: Uso de la información generada para optimizar la compilación usando: -prof_use

Principios hardware del paralelismo

Según el hardware tenemos una primera división de los sistemas multiprocesador:

- de memoria compartida : las cpus pueden acceder cualquier posición de la memoria global del ordenador
- de memoria distribuida : cada cpu tiene su propia memoria y no puede acceder a la memoria del resto de cpus

En general se tiene una combinación:

- Tenemos un ordenador formado por blades (sistema de memoria distribuida) de forma que cada blade tiene cuatro cores con una memoria local (sistema de memoria compartida).
- Por tanto el máximo nivel de paralelismo con threads o con compartición de memoria será de cuatro, y para hacer trabajar mas cores en un problema habrá que usar algún mecanismo de paralelización.

Mecanismos de paralelización

Veamos los tres principales:

- Granja de tareas (Task farming) (Array jobs)
- SPMD (Single Program Multiple Data)
- Data pipelining

Task-farming (I)

- Consiste en dos tipos de entidades:
  - Entidades maestro :
    - Se encargan de descomponer el problema en trozos pequeños y repartirlo entre los procesos esclavos.
    - Además al final pueden recoger y procesar los resultados, para general un resultado final.
  - Entidades esclavo :
    - Reciben una tarea, la ejecutan y devuelven el resultado.
- Es muy útil cuando se tienen múltiples parámetros para un mismo programa.
  - Por ejemplo: distintos puntos de inicio en una simulación.

Task-farming (II)

La forma mas sencilla y óptima de aprovechar un sistema multiprocesador de memoria distribuida consiste en lanzar simultaneamente distintas ejecuciones del mismo o distintos programas por parte de uno o varios usuarios.

Ventajas:

- el aumento de velocidad es real, si se lanzan n ejecuciones simultáneamente se obtendrá el resultado final en el mismo tiempo que si se hace una sola ejecución.

Inconvenientes:

- no aumenta la velocidad de una sola ejecución.

¿ como se hace eso ?

- Usando el sistema de colas
- Ver las transparencias 24 y 25.

Single Program Multiple Data

Es el mas común.

Los datos se reparten entre los distintos procesadores, y el mismo trozo de código se ejecuta en todos, pero sobre distintos trozos de los datos.

Data pipelining

Se basa en una descomposición funcional del problema.

Se identifican las tareas a ejecutar y cada procesador ejecuta una parte del algoritmo global

Se conoce también como paralelismo en el flujo de datos.

Se suele usar en procesamiento de imágenes, siendo cada filtro uno de los pasos del pipeline.

La realidad del paralelismo

Tenemos tres casos principales:

- El programa a usar es un desarrollo externo:
  - el paralelismo está implícito dependiendo de como esté programado.
  - Se puede ejecutar múltiples veces con distintos datos de entrada.
- Si ha sido desarrollado “en casa”
  - Si es paralelo es perfecto, no hay ningún problema.
  - Si no, habría que rediseñarlo para aprovechar el paralelismo que puedan tener los algoritmos, si es que realmente se puede paralelizar.
    - Para ello es recomendable usar una librería que facilite esa tarea, la mas usada es la MPI.

Procedimiento genérico para paralelizar un programa

Cada problema tendrá un procedimiento único para paralelizarlo, generalmente no es sencillo.

En general uno de los procedimientos mas comunes consiste en :

- dividir el problema en varias partes
  - si consiste en operar sobre una matriz, dividirla en varias
- enviar cada una de las partes a un procesador
- si hacen falta mas datos pedirlos al proceso que los tenga
- finalmente reunir los resultados y generar el resultado final.

Introducción al MPI

MPI (Message Passing Interface) es un interfaz de paso de mensajes usado para comunicar distintas partes de una misma aplicación.

Las ventajas de MPI son su escalabilidad, su rendimiento y su portabilidad.

MPI especifica mas de 500 funciones de forma independiente a los lenguajes y al hardware.

Funciones básicas del MPI

Entrando y saliendo:

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

¿ quién soy yo ?

```
int myrank; // contiene el rango de este proceso
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

¿cuantos somos ?

```
int nprocs;  // número de procesos en total
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

Enviando y recibiendo datos en MPI

Para recibir datos de otro nodo:

MPI_Recv(buffer, maxcount, datatype, source, tag,MPI_COMM_WORLD, &status);

source indica el rango del proceso del que queremos recibir los datos

Para enviar datos a otro nodo:

MPI_Send(buffer, count, datatype, destination, tag, MPI_COMM_WORLD);

destination indica el rango del proceso al que queremos enviar los datos

Veamos un ejemplo a continuación

compilando y ejecutando con MPI

Para compilar un programa que use MPI se usa mpicc.

mpicc se encarga de llamar al compilador de gnu o de intel según se defina la variable MPI_CC

Para ejecutar hay que enviarlo al sistema de colas, y allí se le indica cuantas cpus va a usar. En todos los procesadores se ejecutará el mismo programa.

Así se puede probar facilmente con distintos números de cpus para ver como va mejor.

Importante:

El número de cpus óptimos a usar suele depender del tamaño del problema.

Por tanto no por poner muchas mas acabará antes.

Ejemplo de MPI

Veamos a continuación un programa que crea n procesos (el número exacto se indica en el fichero submit del sistema de colas).

Primero el proceso principal (rango 0) escribe un mensaje a cada uno diciendole hola y espera a que le contesten.

Depués estos reciben el mensaje y contestan indicando que están disponibles para trabajar.

Ejemplo de código MPI

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  char idstr[32];  char buff[128];
  int numprocs;  int myid;  int i;

  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,
&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,
&myid);

  if(myid == 0)
  {
    printf("WE have %d processors\n",
numprocs);
    for(i=1;i<numprocs;i++)
    {
      sprintf(buff, "Hola %d", i);
      MPI_Send(buff, 128, MPI_CHAR, i, 0,
MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
    {
      MPI_Recv(buff, 128, MPI_CHAR, i, 0,
MPI_COMM_WORLD, &stat);
      printf("%s\n", buff);
    }
  }
  else
  {
    MPI_Recv(buff, 128, MPI_CHAR, 0, 0,
MPI_COMM_WORLD, &stat);
    sprintf(idstr, " Procesador %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "disponible para trabajar\n");
    MPI_Send(buff, 128, MPI_CHAR, 0, 0,
MPI_COMM_WORLD);
  }

  MPI_Finalize();
}
```

Envío de procesos: universo MPI

Si tenemos programas que usen MPI hay que usar un script especial (mp1script) como ejecutable, e indicar como argumento en el fichero de submit el nombre de nuestro ejecutable.

Hay que crear un fichero de submit para enviarlo.

A los programas que usan MPI hay que indicarles el número de procesos que se crearán, con la variable machine_count del fichero de submit.

Después condor se encargará de distribuirlos entre las cpus disponibles.

fichero de submit para mpi

```
universe = parallel

initialdir = /home/curso01/pruebas/cpi

executable = ./mp1script

arguments = /home/curso01/pruebas/cpi/cpi

log = cpi.log

output = cpi_par_$(PROCESS).output

machine_count = 20

should_transfer_files = yes

when_to_transfer_output = on_exit

WantIOProxy = true

queue
```

número de cpus que usará el proceso mpi

son necesarias para el funcionamiento a nivel interno del condor, debido a transferencias de datos para su funcionamiento

Está en **/home/curso01/pruebas/cpi/**

Copiar el directorio y modificar para usar en la cuenta de cada asistente (modificar el initialdir y arguments)

Envio de procesos MPI

Tenemos un ejemplo en:

ejemplos/compiladores/mpi

Se puede enviar con:

condor_submit hello_mpi.sub

con **condor_q** se pueden ver los que están en ejecución

cuando termine se verá en el fichero de salida en que servidor se ejecutó cada uno de sus procesos.

envío de procesos grandes

Si un proceso ocupa mucha memoria es conveniente indicarlo con ImageSize en el fichero de submit.

Si no se indica el condor va mirando cuanta memoria va ocupando y adaptando el sistema al uso real.

El problema es que si se piden simultaneamente muchas ejecuciones que ocupen mucha memoria todas irán creciendo a la vez y llenarán la memoria todas a la vez, con lo que algunas terminarán dando error de falta de memoria.

Detección de inanición de procesos

Al hacer condor_q podemos ver en el campo SUBMITTED cuando se envió un trabajo al sistema de colas

Si un trabajo lleva mucho tiempo encolado y hay cpus libres hay que mirar si hay algún problema con ese trabajo.

Se puede usar la opción analyze para ver el motivo por el que no pasa a ejecutarse :

condor_q -analyze id_de_trabajo

Usar condor_rm para borrarlo si es necesario