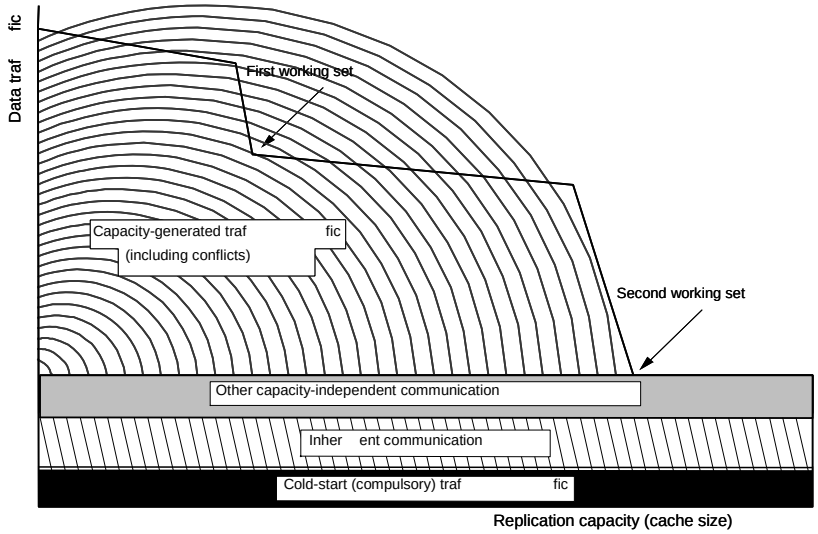
UNIVERSIDAD CARLOS III DE MADRID
DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA.
ARQUITECTURA DE COMPUTADORES II
19 de septiembre de 2007

Para la realización del presente examen se dispondrá de **2 horas y media. NO** se podrán utilizar libros ni apuntes.
Entregar cada ejercicio en hojas separadas.

Ejercicio 1 (1.5 puntos)

La siguiente figura muestra el modelo que relaciona el tamaño de la caché con el tráfico exterior a la caché. ¿Qué elemento novedoso introduce respecto a una arquitectura clásica mono-procesador? ¿Qué característica tiene dicho elemento novedoso?.



SOLUCIÓN:

En este modelo un sistema paralelo se considera una extensión del modelo secuencia para el que su sistema de jerarquía de memoria incorpora uno nuevo nivel (denominado inherent communication) correspondiente a posiciones de memoria no locales a cada procesador. Peticiones a datos no locales tienen asociados comunicaciones este los procesadores, los cuales originan tráfico externo a la memoria caché.

La característica diferencial es que el tráfico asociado es independiente del tamaño de la caché, guardando relación únicamente con el modo en que el programa paralélelo ha sido diseñado.

Ejercicio 2 (1.5 puntos)

¿Qué es la granularidad de un programa paralelo? ¿Qué consideraciones hay que realizar con la misma con miras a obtener un buen rendimiento en la ejecución paralela del programa? Justifica tu respuesta.

SOLUCIÓN:

La granularidad se define como la cantidad de trabajo asociado a cada tarea (en las que se descompone el

programa paralelo).

De forma general, una granularidad elevada hace que el problema se descomponga en pocas tareas, lo cual puede dar origen a problemas de desbalanceo de la carga computacional.
En el caso contrario (granularidad pequeña) por lo general implica un mayor número de comunicaciones y sincronizaciones, lo cual tampoco en beneficioso, dado que incrementa el peso de estas operaciones sobre el tiempo total de ejecución.

Por lo general se procura obtener una granularidad suficientemente grande como para poder distribuir la carga sobre los procesadores. Es decir, la mayor posible sin que se originen problemas de desbalanceo.

Ejercicio 3 (2 puntos)

Un computador paralelo brinda la operación atómica ***fetch&inc(v)*** (capta e incrementa) que capta la variable **v** e incrementa en **1** su valor. La operación ***fetch&incr*** consume 100 ciclos de reloj.

1. Escriba el código necesario para implementar una barrera utilizando la operación ***fetch&inc(v)*** y *n* procesadores.
2. Determine el número de ciclos necesarios para que 10 procesadores atraviesen la barrera.

Nota:
Asuma que cada transacción en el bus (fallo de lectura o de escritura) consume 100 ciclos de reloj.
Ignore el tiempo de lectura o escritura de las variables en la caché, así como también el tiempo de otras operaciones (y acceso a variables) no relacionadas con la sincronización.

Solución

Parte 1
Variable compartida: "espera" con valor inicial: 1. Mientras "espera" valga "1" el proceso no puede abandonar la barrera (i.e. al menos un proceso no ha alcanzado la barrera).
Variable compartida: "contador" con valor inicial: 0. Esta variable lleva cuenta del número de procesos que han alcanzado la barrera.

```
fetch&inc(contador);
if (contador == 10) {

  contador = 0;
  espera   = 0;
}
else {
  while (espera == 1) ;
}
```

Parte 2
Para *n* procesadores, esta implementación requiere *n* operaciones fetch&inc. Cada operaci de fetch&inc supone n lecturas de la variable contador (originando transacciones a través del bus) y n escrituras de la variable (originando también transacciones a través del bus). Adicionalmente existen *n* fallos de caché para acceder a la variable "espera", las cuales originan n transacciones en el bus. Para 10 procesadores esto son 30 transacciones de bus, o 3000 ciclos de reloj.

Ejercicio 4 (2.5 puntos)

Sea una imagen de 4 Megapíxeles (2048x2048) codificada como 256 niveles de gris y almacenada en una matriz (denominada imagen[2048][2048]) a la que se desea extraer el histograma y la media de los valores

de sus píxeles.

1. Proponga un algoritmo en C paralelizado mediante directivas OpenMP que calcule dichas características y proporcione el mayor grado de paralelismo posible. Explica de forma justificada el por qué de cada directiva o cláusula OpenMP utilizada.
2. Explique en qué arquitecturas resulta recomendable, desde el punto de vista del rendimiento, paralelizar código usando OpenMP.

Nota:
El histograma de una imagen indica qué número de píxel tienen un determinado valor. Es decir, cuántos píxeles tienen un nivel 0 de gris, cuántos tienen un nivel 1 de gris, etc.

1.- Una posible solución, bastante simple además, es la que se muestra a continuación:

```
int main()
{
  int val,suma;
  int imagen[2048][2048];
  int histo[256];
  double media;
  …
   media=0;
   suma=0;
  <inicialización de arrays>
  <lectura de la imagen sobre el array>(imagen)

  #pragma omp parallel for  private (y, val) reduce (+ : suma)
  for (x=0; x<2048; x++)
    for (y=0;y<2048;y++)
    {
      val= imagen[x][y];
      suma=suma+val;
  #pragma omp atomic
      histo[val]++;
    }
  media=suma/(2048*2048);

  <resto del programa>
…
}
```

2.- En primer lugar y por regla general, sólo tiene sentido usar OpenMP si el código se va a ejecutar en una máquina multiprocesador, como pueden ser los cada vez más populares sistemas distribuidos de memoria compartida y multi-core (aunque también sobre sistemas monoprocesadores con aplicaciones donde el multithreading solape cálculo de CPU y E/S). En segundo lugar, hay que cuidar que el speedup previsto compense el coste en la creación de los distintos threads.

Problema 5 (2.5 puntos)

P0	P1	P2	P3	P4	P5	P6	P7
Computación	Computación	Computación	Computación	Computación	Computación	Computación	Computación
Send(P1)	Send(P3)	Send(P0)	Send(P2)	Send(P5)	Send(P7)	Send(P4)	Send(P6)
Recv(P2)	Recv(P0)	Recv(P3)	Recv(P1)	Recv(P6)	Recv(P4)	Recv(P7)	Recv(P5)

La tabla superior representa un paso de ejecución de un programa paralelo, el cual consiste de una fase de cómputo y una de comunicaciones. La primitiva Send(Pi) envía 1024 bytes de datos del procesador actual al procesador Pi. La primitiva Recv(Pj) recibe 1024 bytes de datos del procesador Pj.

Se desea diseñar un sistema de memoria distribuida que optimice las comunicaciones del programa anterior. Se pide:

1) ¿Qué topología de red de comunicaciones es más adecuada: fat-tree o hipercubo?. Justifica tu respuesta.
2) Asumiendo que la estrategia de enrutamiento es la *cut-through,* el routing delay es 2 µs, el ancho de banda de la red es 1024Mbytes/s y el overhead de comunicaciones es 2 µs. Calcular el tiempo de comunicaciones del programa anterior para una red hipercubo y una red fat-tree.

Solución:

1) La red hipercubo es óptima: cada procesador Pi envía data a un procesador Pj para el que la distancia entre los dos es de 1 unidad. En este caso no es necesario un hop de comunicaciones intermedio para realizar en enrutamiento (debido a que no es necesario).

2) Hipercubo: 2 (recv overhead) + 2 (send overhead) + 1024 bytes / 1024 Mbytes/s = 5 µs para el primer intercambio y 5 µs para el segundo. Total: 10 µs

Fat tree: empleando un algoritmo de planificación óptimo el máximo número de hops que requiere una transferencia es de 3. De este modo, tenemos 2 (recv overhead) + 2 (send overhead) + 3 hops * 2 µs/hop + 1024 bytes / 1024 Mbytes/s = 11 µs para el primer intercambio y 11 µs para el segundo. Total, 22 µs.