Ordenamiento en Paralelo en OpenMPI

Mariana Carmona Baez & Omar Díaz Landa *ITAM*

29 de mayo de 2017

El objetivo del proyecto es utilizar la API de OpenMPI para implementar un algoritmo de ordenamiento de arreglos de manera distribuida

1. Introducción

1.1. Ordenamiento

Uno de los procedimientos más comunes y útiles en el procesamiento de datos, es la ordenación de los mismos. El objetivo de ordenar generalmente es facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto.

Ordenar significa reagrupar o reorganizar un conjunto de datos u objetos en una secuencia especifica, la cual puede ser de dos formas distintas: Ascendente o Descendente.

Los algoritmos de ordenación se clasifican en dos categorías:

- Ordenación interna o de arreglos
- Ordenación externa o de archivos

En este trabajo se concentra en la Ordenación interna. Recibe este nombre ya que los elementos o componentes del arreglo se encuentran en la memoria principal de la computadora.

Los métodos de ordenación interna a su vez se clasifican en:

- Métodos directos $O(n^2)$
- Métodos logarítmicos $O(nlog(n))$

Los métodos directos, son los más simples y fáciles de entender, son eficientes cuando se trata de una cantidad de datos pequeña. Los métodos logarítmicos, son más complejos, difíciles de entender y son eficientes en grandes cantidades de datos.

Los métodos directos más conocidos son:

- Ordenación por intercambio:
- Ordenación por inserción:
- Ordenación por selección:

En la siguiente sección se verán a detalle algunos algoritmos de ordenación por intercambio.

1.2. MPI

Los sistemas MIMD (*Multiple instruction, multiple data*) soportan instrucciones simultáneas que operan en múltiples flujos de datos. Por esto, los sistemas MIMD consisten normalmente de una colección de unidades de procesamiento o *cores* totalmente independientes. Además los procesadores son asíncronos, es decir, pueden trabajar a su propio ritmo.

Hay dos tipos principales de sistemas MIMD: los sistemas de memoria compartida (*shared-memory system*) y los sistemas de memoria distribuida (*distributed-memory system*). En un sistema de memoria compartida, una colección de procesadores autónomos conectados a una memoria global a la que todos tienen acceso. En un sistema de memoria distribuída, existen los pares procesador-memoria que se comunican por medio de una red. Por lo tanto, en un sistema de memoria distribuida, los procesadores usualmente se cominican explícitamente por medio de mensajes

(*message-apssing*) o funciones que permitan el acceso a la memoria de otro procesador.

Este trabajo utiliza los sistemas de memoria distribuida textitmessage-apssing. En particular, la implementación que se usa es MPI (*Message-Passing Interface*). MPI no es un lenguaje de programación nuevo, define una librería de funciones que pueden ser llamadas desde programas de `C`, `C++` y `Fortran`.

2. Algoritmos de Ordenación por intercambio

La ordenación por intercambio consiste en comparar dos elementos del arreglo y determinar si existe un intercambio entre ellos, para esto debe definirse el tipo de ordenamiento que se quiere ya sea ascendente o descendente.

Los algoritmos de ordenación directa por intercambio que se verán en la siguiente sección son: *Bubblesort* (método burbuja) y *Quicksort* (ordenamiento rápido)

2.1. Quicksort

El método de ordenamiento rápido o método *quicksort*, es una técnica basada en otra conocida con el nombre "divide y vencerás", que permite ordenar una cantidad de elementos en un tiempo proporcional a n^2 en el peor de los casos o a $nlog(n)$ en el mejor de los casos. El algoritmo original es recursivo, como la técnica en la que se basa.

La descripción del algoritmo para el método de ordenamiento *quicksort* es la siguiente:

1. Elegir uno de los elementos del arreglo al que se llama pivote.
2. Acomodar los elementos del arreglo a cada lado del pivote, de manera que del lado izquierdo del pivote queden todos los menores al pivote y del lado derecho los mayores al pivote. En este momento el pivote ocupa el lugar que le corresponderá en el arreglo ordenado.
3. Con el pivote en su lugar se forman dos subarreglos, uno formado por los elementos del lado izquierdo del pivote y otro por los elementos del lado derecho del pivote.
4. Repetir este proceso de forma recursiva para cada subarreglo mientras estos contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Para elegir un pivote se puede aplicar cualquiera de las siguientes tres opciones:

- El pivote será el primer elemento del arreglo.
- El pivote será el elemento que esta a la mitad del arreglo.
- Que el pivote se elija de entre tres elementos del arreglo (cualesquiera), los cuales se deben comparar para seleccionar el valor intermedio de los tres y considerarlo como el pivote.

La forma o técnica de reacomodo de los elementos del lado izquierdo y derecho del pivote, aplica el siguiente procedimiento que es muy efectivo. Se utilizan dos índices: `izq`, al que se llama índice inicial, y `der`, al que se llama índice final. Conociendo estos elementos el algoritmo queda como sigue,

1. Recorrer el arreglo simultáneamente con `izq` y `der`: por la izquierda con `izq` (desde el primer elemento), y por la derecha con `der` (desde el último elemento).
2. Mientras el arreglo en su posición `izq` (arreglo[`izq`]) sea menor que el pivote, se continúa el movimiento a la derecha.
3. Mientras el arreglo en su posición `der` (arreglo[`der`]) sea mayor que el pivote, se continúa el movimiento a la izquierda.
4. Terminando los movimientos se compara los índices y si izq es menor o igual al `der`, se intercambian los elementos en esas posiciones y las posiciones se cambian `izq` a la derecha y `der` a la izquierda.
5. Repetir los pasos anteriores hasta que se crucen los índices (`izq` sea menor o igual a `der`).
6. El punto en que se cruzan los índices es la posición adecuada para colocar el pivote, porque se sabe que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

2.2. Bubblesort

Este algoritmo realiza el ordenamiento o reordenamiento de una lista a de n valores, en este caso de n términos numerados del 0 al $n-1$; consta de dos bucles anidados, uno con el índice i, que da un tamaño menor al recorrido de la burbuja en sentido inverso de 2 a n, y un segundo bucle con el índice j, con un recorrido desde 0 hasta $n-i$, para cada iteración del primer bucle, que indica el lugar de la burbuja.

La burbuja son dos términos de la lista seguidos, j y $j+1$, que se comparan: si el primero es mayor que el segundo sus valores se intercambian.

Esta comparación se repite en el centro de los dos bucles, dando lugar a la postre a una lista ordenada. Puede verse que el número de repeticiones solo depende de n y no del orden de los términos, esto es, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada.

El algoritmo consiste en comparaciones sucesivas de dos términos consecutivos ascendiendo de abajo a arriba en cada iteración, como la ascensión de las burbujas de aire en el agua, de ahí el nombre del procedimiento. En la primera iteración el recorrido ha sido completo, en el segundo se ha dejado el último término, al tener ya el mayor de los valores, en los sucesivos se irán dejando de realizar las íltimas comparaciones.

Debido a la naturaleza de este algoritmo que requiere un orden secuencial de las comparaciones, no tiene sentido paralelizarlo.

2.2.1. Odd-even transposition sort

Una variante del *Bubblesort* es el *Odd-even transposition sort*, el cual es posible paralelizar. La idea se basa en disociar o desacoplar las comparaciones e intercambios. El algoritmo consiste en una secuencia de **fases** de dos tipos diferentes. Durante las fases *pares* se realiza *Bubblesort* en los valores índices pares, $(0,1),(2,3),(4,5),\ldots$, y durante las fases *impares* en los valores con índices impares $(1,2),(3,4),(5,6),\ldots$.

En general, el algoritmo garantiza que se pueden ordenar una lista n valores en a lo más n fases. La prueba más común usa el **Lema de ordenamiento de 0-1**, lo que permite restringir el problema a únicamente ceros y unos.

Lema 1. *Ordenamiento de 0-1. Si un algoritmo de ordenamiento por intercambio ordena una secuencia de ceros y unos, entonces ordena cualquier secuencia arbitraria.*

Demostración. Se empieza la prueba en dirección contraria, es decir, si no ordena una secuencia arbitraria entonces no ordena una secuencia de ceros y unos. Asumir que se tiene una secuencia $a = a_1,\ldots,a_n$ que no está ordenada de manera correcta. Existe un valor más pequeño k tal que el valor en el nodo v_k es estrictamente más grande que el k-ésimo valor más pequeño $a(k)$. Definir una entrada $x_i^* = 0 \Leftrightarrow x_i \leq x(k), x_i^* =$ 0e.o.c Cuando el algoritmo compara una pareja de unos y ceros, no es importante si intercambia los valores o no, simplemente se asume que hace lo mismo que con la entrada x. Por otro lado, cuando el algoritmo compara los valores $x_i^* = 0$ y $x_j^* = 1$ esto significa que $x_i \leq x(k) < x_j$. Por lo tanto, en este caso la operación de comparar-intercambiar hará lo mismo en ambas entradas. Se concluye que el algoritmo va a ordenar x^* de la misma manera que x, es decir, la salida con únicamente ceros y unos tampoco será correcta. □

Teorema 1. *El algoritmo Odd-even sort con n fases, ordena toda secuencia de longitud n*

Demostración. Gracias al lema anterior, solamente se tiene que considerar un arreglo de ceros y unos. Sea j_1 el nodo con valor 1 con el índice más alto. Si j_1 es impar (par) se moverá en el primer (segundo) paso. En cualquier caso, se moverá a la derecha en cada paso siguiente hasta alcanzar el nodo más a la derecha v_n. Sea j_k el nodo con el k-ésimo 1 más a la derecha. Se muestra por inducción que j_k no está bloqueado ya que se mueve hasta que encuentra su destino después de la fase k. Ya tenemos anclada la inducción en $k = 1$. Como j_{k-1} se mueve después de la fase $k-1$, j_k obtiene un cero a la derecha en cada fase después de la fase k.

□

En la siguiente sección se implementa este algoritmo en paralelo utilizando *OpenMPI*.

3. *Parallel odd-even transposition sort*

Para diseñar un programa en paralelo no existe un proceso mecánico que se pueda seguir, de ser así, se

podría paralelizar cualquier programa secuencia. Sin embargo, Ian Foster (1995) propuso un esquema que consiste en lo siguiente:

1. **Particionar:** Dividir el cómputo a realizar y los datos operados por el programa en pequeñas tareas. Lo importante es la identificación de tareas que se pueden ejecutar en paralelo.
2. **Comunicación:** Determinar qué comunicación debe llevarse a cabo entre las tareas identificadas en el paso anterior.
3. **Aglometación y Agregación:** Combinar las tareas y comunicaciones definidas en el primer paso en tareas más grandes. Por ejemplo, si una tarea A tiene que ser ejecutada antes de que la tarea B se puede ejecutar, entonces se deben de agregar en una sola tarea.
4. **Mapeo:** Asignar las tareas compuestas que se identificaron en el paso anterior a procesos/*threads*. Esto se hace con el fin de que la comunicación se minimice y cada proceso/*thread* tenga aproximadamente la misma cantidad de trabajo.

Si se tiene un total de n objetos y p procesos, el algoritmo va a empezar y terminar con n/p objetos asignados a cada proceso, se asume que n es divisible por p. En un inicio, no hay restricciones respecto de qué objetos se asignan a qué proceso. Sin embargo, cuando el algoritmo termina, se tiene lo siguiente:

- Los objetos asignados a cada proceso deben de estar ordenados de manera ascendente.
- Si $0 \leq q < r < p$, entonces cada objeto asignado al proceso q tiene que ser menor o igual a cada objeto asignado al proceso r.

Así, si se alinean los objetos de acuerdo al rango del proceso, es decir, objetos del proceso 0 primero, después objetos del proceso 1, y así sucesivamente, entonces los objetos deberían de estar ordenados de manera ascendente. En este trabajo se asume que los objetos son enteros (`int`).

Es natural paralelizar *odd-even transposition sort* dado que todas las comparaciones e intercambios de una fase pueden ocurrir de manera simultánea. Para lograr esto, se aplica de la siguiente manera la metodología de Foster.

- Tareas: Determinar el valor de a[i] al final de la fase j.
- Comunicaciones: La tarea que está determinando el valor de a[i] necesita comunicarse, ya sea con quien está determinando el valor de a[$i-1$] o de a[$i+1$]. Además, el valor de a[i] al final de la fase j tiene que estar disponible para determinar el valor de a[i] al final de la fase $j+1$

Recordar que cuando el algoritmo de ordenamiento empieza y termina su ejecución, a cada proceso le es asignado n/p objetos. En este caso la agregación y el mapeo están parcialmente especificados por la descripción del problema. Si $n = p$, dependiendo de la fase, el proceso i puede enviar su valor actual a[i], ya sea al proceso $i-1$ o al proceso $j+1$. Al mismo tiempo, debe de recibir el valor que está guardado en $i-1$ o $j+1$, respectivamente, y entonces asignar el valor que debe de estar guardado en a[i] para la siguiente fase.

Sin embargo, es poco probable que se quiera aplicar el algoritmo cuando $n = p$, dado que es improbable que se tenga a nuestra disposición miles de procesadores, cuando además ordenar miles de valores es un problema trivial para un solo procesador. Más aún, incluso cuando se cuenta con miles o millones de procesadores, el costo de enviar y recibir mensajes para cada comparación e intercambio retarda el proceso de tal manera que no será útil. El costo de comunicación, en la mayoría de los casos, es más alto que el costo de las operaciones locales.

Referencias

Figueredo, A. J. y Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20(1):317–330.

McNeil, Alexander J., Frey, Rüdiger y Embrechts, Paul (2015). *Quantitative Risk Management: Concepts, Techniques and Tools*, Princeton University Press, New York.