Programación Paralela

Jesús Manuel Mager Hois

8 de marzo de 2013

Índice

1. Introducción **3**

2. Características **3**
2.1. Equipos multicore 3
2.2. Memoria compartida 4
2.3. OpenMP 4

3. Consideraciones sobre C **5**
3.1. API de OpenMP para C 5
3.1.1. Directivas 5
3.1.2. Biblioteca de rutinas de tiempo de ejecución 10
3.2. Sistema de producción multiplataforma (Autoconf y OpenMP) 14

4. Manos a la obra **17**
4.1. Primer programa en OpenMP (directiva parallel) 17
4.2. Directiva for 18
4.3. Directiva section 21
4.4. Directiva single 22
4.5. Directiva barrier 23
4.6. Directiva critical 24
4.7. Directiva atomic 25

5. Implementación **26**
5.1. Implementación de búsqueda secuencial 26

1. Introducción

La programación paralela es una forma de computación donde varios cálculos son efectuados de manera simultánea, bajo el principio de que cada problema puede ser separado en partes para su tratamiento. Este tiempo de programación es válido para equipos que tienen procesadores con varios núcleos (multicore) y computadoras con varios procesadores de memoria compartida. Sin embargo, se ha agregado un nuevo nivel de dificultad a la programación, donde las sincronización entre las subtareas y las condiciones de carreras introducen nuevos tipos de errores en los programas y nuevos retos para realmente obtener buenos resultados en ciertos problemas. También es importante considerar la llamada ley de Amdahl donde se plantea que los programas paralelos tienen un límite de mejoramiento marcado por las partes del mismo que no pueden ser paralelizados.

Para implementar este tipo de programación se ha implementado librerías que facilitan el trabajo paralelo, como son POSIX Threads, OpenMP y MPI (Message Passing Interface). Sin embargo también existen lenguajes de programación que implementan de manera automática la paralelización, pero con resultados limitados. Ejemplos de estos lenguajes de paralelizado implícito es Parallel Haskell.

En este breve texto, trataremos el lenguaje de programación C con la librería de OpenMP. Presentaremos el API de OpenMP, así como ejemplos sencillos para que el lector pueda acercase a la programación paralela. Con ejemplos dos ejemplos más complejos presentamos también aplicaciones reales de la programación en paralelo para el mejoramiento de algoritmos conocidos.

2. Características

2.1. Equipos multicore

Un equipo multicore es un procesador en un único componente con dos o más unidades centrales de procesamiento llamados cores o núcleos, que son procesadores que ejecutan instrucciones. Estas instrucciones son instrucciones típicas de cada procesador, como puede ver add, mov, etc... que pueden ser ejecutadas simultáneamente por sus diferentes núcleos. Al llegar la computación a un nivel límite de ciclos por segundo en sus procesadores, las grandes compañías productoras de procesadores han optado por incrementar el número de núcleos de sus procesadores. Los primeros procesadores con estas características fueron el AMD Phenom II X2 y Intel Core Duo. Sin embargo para poder ganar realmente sacar provecho de las tecnologías es necesario que los algoritmos computacionales utilicen estas características. Tenemos una serie de algoritmos conocidos como embarazosamente paralelos que pueden ganar mayor provecho de esta situación. Independientemente de esto, el presente texto tratará de generar una primer acercamiento a la programación multicore y multiprocesador utilizando la librería OpenMP.

2.2. Memoria compartida

El término Computadoras Paralelas de Memoria Compartida(en inglés *Shared-Memory Parallel Computers* SMPs) fue originalmente creado para designar sistemas multiprocesadores simétricos, computadoras paralelas de memoria compartida donde los procesadores individuales comparten la entrada y salida de la memoria de tal manera que cada uno de ellos puede acceder a la memoria en cualquier locación con la misma velocidad, esto significa que tienen *acceso uniforme a memoria*(en inglés *uniform memory access* UMA). Muchas computadoras de memoria compartida menores son en este sentido simétricas. Sin embargo, computadoras más grandes, usualmente no satisfacen esta definición;incluso si la diferencia de tiempo de acceso es relativamente pequeña, ya que alguna memoria puede encontrarse mas çerca"de uno o más procesadores, y estos logran acceder más rápido a esta memoria. A estas computadoras se le dice que tienen un acceso a *memoria no uniforme coherente-cache*(en inglés *cache-coherent non-uniform-memory access* cc-NUMA).

En la actualidad todos los grandes fabricantes de hardware ofertan algún tiempo de computadoras de memoria compartida, desde dos procesadores, hasta cientos o incluso miles.

2.3. OpenMP

OpenMP es API multiplataforma para multiprocesos de memoria compartida. Está escrito para usarse de manera nativa desde C, C++ y Fortran. Se encuentra disponible para una gran variedad de arquitecturas y sistemas operativos, entre los que podemos destacar Windows, Mac OS X y la familia de sistemas operativos Unix. OpenMP es una estándar y está bajo la administración de un consorcio no lucrativo impulsado por los mayores productores de hardware.

Entre los miembros de la Mesa de Revisión de Arquitectura, que es el consorcio del estándar se encuentran miembros permanentes: AMD, CAPS-Entreprise, Convey Computer, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, NVIDIA, Oracle Corporation, The Portland Group, Inc. y Texas Instruments. Además se tiene ta bién miembros auxiliares: ANL, ASC/LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH Aachen University, Texas Advanced Computing Center.

Se implementa el multihilo partiendo de un hilo maestro, que con una serie de instrucción son partidas creando hilos esclavos, donde las tareas son distribuidas a estos hilos. Los hilos son ejecutados entonces y ejecutados en tiempo de ejecución a diferentes procesadores. Una vez los hilos hallan terminado sus tareas paralelizadas, los hilos vuelven a unirse otra vez en el hilo maestro.

3. Consideraciones sobre C

3.1. API de OpenMP para C

3.1.1. Directivas

Una directiva ejecutable de OpenMP aplica al bloque de construcción OpenMP. Un bloque estructurado es una declaración simple o una declaratoria compuesta con una sola entrada al principio y una sola salida al final.

1. **Parallel**

   La construcción **parallel** crea un equipo de hilos y comienza con la ejecución paralela.

```
#pragma omp parallel [clausula[[,]clausula]...]
        bloque estructurado
clausulas:
        if(expresion escalar)
        num_threads(expresion de enteros)
        default(shared | none)
        private(lista)
        shared(lista)
        copyin(lista)
        reduction(operador:lista)
```

2. **Loop**

   La construcción **loop** especifica que las iteraciones de los ciclos serán distribuidas y ejecutadas entre el equipo de hilos.

```
#pragma omp for [clausula[[,]clausula]...]
        ciclos-for
clausula:
        private(lista)
        firstprivate(lista)
        lastprovate(lista)
        reduction(operador: lista)
        schedule(tipo[,extencion del trabajo])
        collapse(n)
        ordered
        nowait
```

   Tipo:

   - **static**: Las iteraciones son divididas en partes de un tamaño definido. Las partes son asignadas a los hilos del equipo en una forma de round-robin según el número de hilos.
   - **dynamic**: Cada hilo ejecuta una cantidad de iteraciones, y posteriormente pide otra cierta cantidad hasta que no quede ninguna parte del ciclo.

- **guided**: Cada hilo ejecuta una parte de iteraciones, después pide otra parte, hasta que no quede ninguna parte sin asignar. El tamaño de las partes del siclo comienzan grandes, y posteriormente se irán reduciendo hasta llegar al indicado *chunk_size*.
- **auto**: La decisión de como delegar las tareas se relega al compilador o al sistema de tiempo de ejecución.
- **runtime**: El trabajo y el tamaño de las partes a procesar será decidido en tiempo de ejecución.

```
for(var = lb; var relational-op b; var+=incr)
```

3. **Sections**

   La construcción de secciones contiene una serie de bloques estructurados que serán distribuidos y ejecutados entre un equipo de hilos.

```
#pragma omp sections [clausula[[,]clausula]...]
        {
        [#pragma omp section]
                bloque estructurado
        [#pragma omp section]
                bloque estructurado
        ...
        }
clausulas:
        private(lista)
        firstprivate(lista)
        lastprovate(lista)
        reduction(operador: lista)
        nowait
```

4. **Single**

   La construcción single especifica que el bloque estructurado asociado es ejecutado únicamente por uno de los hilos que componen el equipo de hilos(no necesariamente el hilo maestro), en el contexto de una tarea implícita.

```
#pragma omp single [clausula[[,]clausula]...]
        bloque estructurado

clausulas:
        private(lista)
        firstprivate(lista)
        copyprivate(lista)
        nowait
```

5. **Parallel Loop**

La construcción de ciclo paralelo (parallel loop) es una abreviación para especificar una construcción paralela que contiene uno o más ciclos, sin otras declaraciones.

```
#pragma omp parallel for [clausula[[,]clausula]...]
        ciclo for

clausulas:
        Cualquiera aceptada por las directivas for o
        parallel, excepto nowait, con significados
        exactamente iguales.
```

Ejemplo de Parallel Loop:

```
void simple(int n, float *a, float *b)
{
         int i;
#pragma omp parallel for
         for (i=1; i<n; i++)
                 b[i] = (a[i] + a[i-1]) / 2.0;
}
```

6. **Parallel Sections**

La construcción de secciones paralelas (**parallel sections**) es una abreviación de especificar la construcción **parallel** conteniendo una constricción **section** y ninguna otra declaración.

```
#pragma omp parallel sections [clausula[[,]clausula]...]
        {
        [#pragma omp section]
                bloque estructurado
        [#pragma omp section]
                bloque estructurado
        ...
        }
clausulas:
        Cualquiera aceptada por las directivas sections o
        parallel, excepto nowait, con significados
        exactamente iguales.
```

7. **Task**

La construcción task define explicitamente una tarea. Los datos del entorno de la tarea son creados de acuerdo a los atributos de la clausula de compartir datos en la construcción task y por los que por defecto se aplican.

```
#pragma omp task [clausula[[,]clausula]...]
        blque estructurado
clausula:
        if(expresion escalar)
        final(expresion escalar)
```

```
        united
        default(shared \| none)
        mergeable
        private(lista)
        firstprivate(lista)
        lastprivate(lista)
        shared(lista)
```

8. **Taskyield**

   La construcción **Taskyield** especifica que la actual tarea puede ser suspendida en favor de la ejecución de otra tarea diferente.

```
#pragma omp taskyield
```

9. **Master**

   La construcción **master** especifica un bloque estructurado que es ejecutado por el hilo maestro del equipo de hilos. No existe ninguna barrera implícita ya sea de entrado o de salida al hilo maestro.

```
#pragma omp master
```

10. **Critical**

    La construcción **critical** restringe la ejecución del bloque estructurado asociado a un solo hilo en un momento.

```
#pragma omp critical [(nombre)]
        bloque estructurado
```

11. **Barrier**

    La construcción **barrier** especifica una barrera explícita en el punto en el cual la construcción aparece.

```
#pragma omp barrier
```

12. **Taskwait** La construcción taskwait especifica una pausa en la competencia de tareas hijas con la tarea actual.

```
#pragma omp taskwait
```

13. **Atomic**

    La construcción **atomic** asegura que una locación específica es actualizada automáticamente, en vez de exponerla a posibles hilos que escriban sobre el de manera múltiple o simultáneo.

```
#pragma omp atomic [read | write | update | capture ]
   expresion-sentencia
#pragma omp atomic capture
   bloque estructurado
```

Donde la expresión-sentencia puede tener una de las siguientes formas:

Si *clause* es...	**expresión-sentencia:**
read	*v = x;*
write	*x = expr;*
update o en caso de no estar presente	*x++; x–; ++x; –x; x binop = expr; x = x binop expr;*
capture	*v=x++; v=x–; v=++x; v=–x; v=x binop=expr;*

y el *bloque estructurado* puede tener una de las siguientes formas:

{v=x; x binop = expr;} {x binop = expr; v=x;} {v=x; x binop expr;} {x=x binop expr; v=x;} {v=x; x++;} {v=x; ++x;} {++x; v=x;} {x++; v=x;} {v=x; x++;} {v=x; –x;} {–x; v=x;} {x–; v=x;}

14. **Flush** La estructura **flush** ejecuta la operación OpenMP flush, que hace a la vista temporal de memoria consistente con la memoria, y fuerza un orden sobre la operaciones de memoria de las variables.

```
#pragma omp flush [(list)]
        bloque estructurado
```

15. **Ordered**

    La construcción **ordered** especifica un bloque estructurado en una región de ciclo que será ejecutada en el orden de las iteraciones del ciclo. Esto sequencializa y ordena el código dentro de una región ordenada mientras que permite al código que se encuentra fuera de la región correr en paralelo.

```
#pragma omp ordered [(list)]
        bloque estructurado
```

16. **Threadprivate**

    La directiva **threadprivate** especifica que las variables en la lista serán replicados, donde en cada hilo tendrá su propia copia.

```
#pragma omp threadprivate [(list)]
```

list: Una lista separada por comas de variables del ámbito del archivo, del namespace o del bloque estático, que son variables de tipos completos.

3.1.2. Biblioteca de rutinas de tiempo de ejecución

- Rutina de entorno de ejecución

  Las rutinas de entorno de ejecución afectan y monitorean hilos, procesos y el entorno paralelo en general.

  void omp_set_num_threads (int *num_threads*);

  Afecta el número de hilos usados por la regiones paralelas subsecuentes que no especifican un clausula num_threads.

  int omp_get_num_threads(void);

  Regresa el número de hilos de equipo actual.

  int omp_get_max_threads(void);

  Regresa el número máximo de hilos que pueden ser usados por un nuevo equipo que use la construcción parallel sin la cláusula **num_threads**.

  int omp_get_thread_num(void);

  Regresa la identidad del hilo en que se encuentra donde la identidad tiene un rango de 0 al tamaño del equipo de hilos menos 1.

  int omp_get_num_procs(void);

  Regresa el número de procesadores disponibles para el programa.

  int omp_in_parallel(void);

  Regresa **verdadero** si la llamada a la rutina se realiza en una región **parallel**; de lo contrario regresa **falso**.

  void omp_set_dynamic(int *dynamic_threads*);

  Habilita o deshabilita el ajuste dinámico del número de hilos al asignar el valor a la variable *dyn-var* ICV.

  int omp_get_dynamic(void);

  Regresa el valor de *dyn-var* IVC, determinando si el ajuste dinámico del número de hilos se encuentra habilitado o deshabilitado.

  void omp_set_nested(int *nested*);

  Habilita o deshabilita paralelismo anidado asignando ICV a la *nest-var*.

  int omp_get_nested(void);

  Regresa el valor de la *nest-var*, la cual determina si el paralelismo anidado se encuentra activado o desactivado.

  void omp_set_schedule(omp_sched_t *kind*, int *modifier*);

  Afecta el itinerario que es aplicado cuándo el **tiempo de ejecución** es usado como tipo de itinerario, estableciendo el valor de *run-sched-var* ICV.

kind es uno de **static, dynamic, guided, auto** o una implementación definida de itinerario.

Véase la descripción de la construcción for.

void omp_get_schedule(omp_sched_t **kind*, int **modifier*);

Regresa el valor de *ru-scher-var* ICV, el cual es el itinerar¡o aplicado cuándo el itinerario de

tiempo de ejecución es usado.

Véase *kind* arriba.

int omp_get_thread_limit(void);

Regresa el valor de la variable *thread-limit-var*, que es el número máximo de hilos OpenMP disponibles para el programa.

void omp_set_max_active_levels(int *max_levels*);

Limita el número de regiones **parallel** anidadas, asignando valor a la variable *max-active-levels-var*.

int omp_get_max_active_levels(void);

Regresa el valor de la variable *max-active-levels-var*, la cuál determina el máximo número de regiones

parallel anidadas activas.

int omp_get_level(void);

Regresa el números de regiones **parallel** anidadas que encierran la tarea que contiene la llamada.

int omp_get_ancestor_thread_num(int *level*);

Regresa, para un nivel de anidación dado del hilo actual, el número de hilo del antecesor del hilo actual.

int omp_get_team_size(int *level*);

Regresa, para un nivel de anidación del hilo actual, el tamaño del equipo de hilos al cual pertenecía el antecesor del hilo actual.

int omp_get_active_level(void);

Regresa el número de regiones **parallel** anidadas que encierran la tarea que contiene la llamada.

int omp_in_final(void);

Regresa **verdadero** si la rutina es ejecutada en un final o está incluida en una región de tarea; de lo contrario regresa **falso**.

Rutinas de bloqueo

Las rutinas de cerradura.

Las rutinas de cerraduras ayudan a la sincronización con las cerraduras de OpenMP.

void omp_init_lock(omp_lock_t **lock*);

void omp_init_nest_lock(omp_nest_lock_t **lock*);

Estas rutinas inician una cerradura OpenMP.

void omp_destroy_lock(omp_lock_t **lock*);

void omp_destroy_nest_lock(omp_nest_lock_t **lock*);

Estas rutinas se aseguran de que las cerraduras OpenMP no se encuentran inicializadas.

void omp_set_lock(omp_lock_t **lock*);

void omp_set_nest_lock(omp_nest_lock_t **lock*);

Estas rutinas proveen de los medios para establecer una cerradura OpenMP.

void omp_unset_lock(omp_lock_t **lock*);

void omp_unset_nest_lock(omp_nest_lock_t **lock*);

Estas rutinas proveen de los medios para desactivar una cerradura OpenMP.

int omp_test_lock(omp_lock_t **lock*);

int omp_test_netst_lock(omp_nest_lock_t **lock*);

Estas rutinas tratan de establecer una cerradura OpenMP, pero no suspenden la ejecución de la tarea que ejecuta la rutina.

Rutinas de temporalizador

Las rutinas de temporalización proveen un reloj temporalizador portable.

double omp_get_wtime(void);

Regresa el tiempo del reloj que ha transcurrido, en segundos.

double omp_get_wtick(void);

Regresa la precisión del temporalizador usado por omp_get_wtime.

- Cláusulas

  El conjunto de clausulas que son válidas en una directiva en particular es descrito con la misma directiva. La mayoría de las cláusulas aceptan una lista de elementos separados por una coma. Todos los elementos de la lista que aparecen en una cláusula deben ser visibles.

  Cláusulas de atributo de compartición de datos. Las cláusulas de atributos de compartición de datos se aplican únicamente a variables que son usados en una construcción en la cuál la cláusula aparece.

  default(**shared none**)

  Controla los atributos por defecto de variables de datos compartidos que son aludidos en una construcción **task** o **parallel**.

shared(*list*)

Declara uno o más elementos de lista para ser compartidos por tareas generadas por una construcción **task** o **parallel**.

private(*list*)

Declara uno o más elementos de la lista como privados a la tarea.

firstprivate(*list*)

Declara una o mas elementos de lista como privados para una tarea, e inicializa cada uno de ellos con un calor que corresponda al elemento original que tenía cuando la construcción fue encontrada.

lastprivate(*list*)

Declara uno o más elementos de la lista como privados para una tarea implícita, y causa que el elemento correspondiente original se actualice después de terminar la región paralela.

reduction(*operator*:*list*)

Declara una acumulación para los elementos de lista indicados usando la operación de acumulación asociada. La acumulación se realiza en una copia privada de cada uno de los elementos de lista que son combinados con el elemento original.

Operaciones para reduccuón(valores iniciales)	
+ (0)	|
* (1)	^
- (0)	&&
& (0)	||

Clausulas de copiado de datos

Estas cláusulas permiten copias el valor de las variables private o threadprivate de una tarea implícita o hilo a su variable correspondiente en otra tarea implícita o equipo de hilos.

copyin(*list*)

Copia el valor de la variable threadprivate del hilo maestro a threadprivate de cada uno de los otros miembros de hilos de la región **parallel**.

copyprivate(*list*)

Transmite un calor de un dato de ambiente de una tarea implícita al los datos de ambiente de otra tarea implícita que pertenezca a la región **parallel**.

- Variables de entorno

Los nombres de las variables de entorno son escritas en mayúsculas, y los valores que se les asignan son insensibles a mayúsculas o minúsculas.

OMP_SCHEDULE *type[,chunk]*

Asigna el valor de la variable *run-sched-var* para el tamaño y el tipo del itinerario de tiempo de ejecución. Son tipos válidos de itinerarios para OpenMP **static**, **dynamic**, **guided** o **auto**.*chink* es un entero positivo que especifica el tamaño de pedazo.

OMP_NUM_THREADS *list*

Asigna el valor a la variable *nthreads-var* que especifica el número de hilos que serán usados en la regiones **parallel**.

OMP_DYNAMIC *dynamic*

Asigna el valor de la variable *dyn-var* para el ajuste diánmicos de hilos que se usarán en una región **parallel**. Valores válidos para *dynamic* son **true** o **false**.

OMP_PROC_BIND *bind*

Maneja el valor de la variable global *bind-var*. El valor de esta variable de entorno tiene que ser **true** o **false**.

OMP_NESTED *nested*

Maneja la variable *nest-var* para activar o para desactivar el paralelismo anidado. Valores válidos para *nested* son **true** o **false**.

OMP_STACKSIZE *size*[**B K M G**]

Maneja la variable *stacksize-var* que especifica el tamaño de la pila para hilos creados por la implementación de OpenMP. *size* es un entero positivo que especifica el tamaño de la pila. Si las unidades no se especifica, el tamaño de textitsize es evaluado en kilobytes.

OMP_WAIT_POLICY *policy*

Maneja la variable *wait-policy-var* que controla el comportamiento deseable para esperar hilos. Valores para *policy* son **AVCTIVE** (esperando hilos para el consumo de ciclos de procesador mientras espera) y **PASSIVE**.

OMP_MAX_ACTIVE_LEVELS *levels*

Maneja la variable *max-active-levels-var* que controla el máximo número de regiones **parallel** anidadas activas.

OMP_THREAD_LIMIT *limit*

Maneja la variable *thread-limit-var* que controla el máximo número de hilos que participan en un programa OpenMP.

3.2. Sistema de producción multiplataforma (Autoconf y OpenMP)

Antes de comenzar a utilizar OpenMP, crearemos un sistema de producción con Autotools, que nos permitirá compilar nuestro código en nuestro sistema operativo, sin importar

cual sea, y crear una fácil distribución del mismo para ser compilado en un sinfín de plataformas.

En primer lugar es necesario crear una serie de archivos y un dos directorios, pero estos pueden ser más, según las necesidades. En este caso crearemos un sistema de compilación modesto:

```
touch NEWS README AUTHORS ChangeLog
mkdir m4 src
```

El primer archivo con instrucciones será *configure.ac* y debe tener la forma siguiente:

```
AC_INIT(paral,0.1)

AC_PREREQ(2.50)

AM_INIT_AUTOMAKE(foreign)
AC_CONFIG_SRCDIR([src/paral.c])
AC_CONFIG_HEADER([config.h])

AC_PROG_CC
AC_PROG_INSTALL

AC_HEADER_STDC
AC_CONFIG_MACRO_DIR([m4])
LT_INIT

AX_OPENMP(AC_DEFINE(HAVE_C_OMP, 1, [Define if the compiler supports OpenMP]),
CFLAGS="$CFLAGS $OPENMP_CFLAGS"
AC_CHECK_HEADER(omp.h)

AC_CONFIG_FILES([
Makefile
src/Makefile
])

AC_OUTPUT
```

Donde indicamos como debe de proceder el sistema de producción con su propia creación. En *AC_INIT(paral,0.1)* se debe sustituir el paralel por el nombre del programa que nos encontramos desarrollando y la versión 0.1 por la que el usuario desee.

Además también indicamos que será un programa C con *AC_PROG_CC*, que el programa será apto para instalación (*AC_PROG_INSTALL*) y no como librería, que usaremos C estándar(*AC_HEADER_STDC*), y que deberá crearse (a partir de los *Makefile.am*) el *Makefile* en la raíz de nuestro directorio y en la carpeta *src*, donde por orden pondremos nuestro código.

Tenemos una parte donde se utiliza un marco especial guardado en la carpeta m4 que también se debe crear por parte del usuario y donde debe ser guardado el archivo marco. Este archivo podrá ser conseguido desde la siguiente dirección:

http://www.gnu.org/software/autoconf-archive/ax_openmp.html

Crearemos además dos archivos Makefile.am, uno en la raíz y otro en el directorio *src* que también debemos crear. Este es el contenido de los ficheros:

```
ACLOCAL_AMFLAGS= -I m4
SUBDIRS = src
EXTRA_DIST = AUTHORS ChangeLog NEWS README
```

Donde se especifican los archivos a distribuir además del código fuente y el ejecutable, y los directorios a incluir en la distribución. El contenido del *src/Makefile.am*.

```
bin_PROGRAMS = paral

AM_CFLAGS=-Wall -g

paral_SOURCES = paral.c \
                utils.c

EXTRA_DIST = utils.h
```

Donde *bin_PROGRAMS* nos especifica el nombre del ejecutable a crear, *AM_CFLAGS* nos da a conocer las banderas de compilación que queremos que se incluyan en la compilación, *paral_SOURCES* es una lista de archivos fuentes que se incluirán en la compilación del binario paral. *paral*, al inicio de *SOURCES* se debe cambiar por el nombre del ejecutable a crear.

Con esto debemos de obtener:

```
raiz/
|-- AUTHORS
|-- ChangeLog
|-- configure.ac
|-- m4
|   `-- ax_openmp.m4
|-- Makefile.am
|-- NEWS
|-- README
`-- src
    |-- Makefile.am
    |-- paral.c
    |-- utils.c
    `-- utils.h
```

En los archivos que se adjuntan a este documento se incluye un archivo zip con la base del sistema para que usted adapte su programa a el. Es posible agregar la búsqueda de nuevas

bibliotecas a su entorno y así hacer más complejo sus programas. A continuación veremos como se realiza la conclusión del sistema, para compilar y crear el ejecutable.

Es preferible que guarde el árbol, tal cual está en un sistema SVN, GIT, o simplemente en un directorio aparte, ya que los archivos que se generarán a partir de este paso serán automáticos y llenarán el directorio, pero que no deben ser editados.

```
aclocal
autoreconf --install
./configure
make
```

En sistemas de producción en necesario incluir los dos primeros comandos, donde *aclocal* crea una serie de archivos necesarios de información, y *autoreconf –install* genera todos los archivos make y configure de un sistema tradicional *./configure make*. Su sistema puede ser transportado en una tar ball, con el comando:

```
make dist
```

Si uno pretende instalar el programa en todo el sistema Unix, solo basta introducir este otro comando:

```
sudo make install
```

4. Manos a la obra

4.1. Primer programa en OpenMP (directiva parallel)

Comenzaremos con un primer programa, que muestra la forma general en la cual se incluye le cabecera de OpenMP, además de tener un primer bloque paralelizado. Como se podrá ver, este primer programa es muy simple y únicamente obtiene información de OpenMP respecto a la identidad del hilo en el que se encuentra y sobre el total de hilos que se encuentran activos en tiempo de ejecución. Esta información se imprime en pantalla.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int n_hilos, id_hilo;
    omp_set_num_threads(4);
    /*
    * Se generan una serie de hilos. Con la sentencia
    * de procompilado #pragma se indica una sentencia
    * OpenMP.
```

```
    */
#pragma omp parallel private(id_hilo, n_hilos)
    {
        /* Con omp_get_thread_num() se obtiene el número de hilos que
        * se encuentra utilizando. */
        id_hilo = omp_get_thread_num();
        printf("Este es el hilo con identidad: %d\n", id_hilo);

        /* Si la identidad del hilo actual es 0, estamos en el
        * hilo maestro. Una vez que se acabe la secci n paralela
        * todo se volverá a fundir en este hilo.*/
        if (id_hilo == 0)
        {
            n_hilos = omp_get_num_threads();
            printf("Cantidad total de hilos activos: %d\n", n_hilos);
        }
    }
    /* Se termina el bloque paralelo. */
}
```

La cabecera *omp.h* declara las variables, funciones y sentencias para el uso de OpenMP. Todo programa que utilice OpenMP debe incluir esta cabecera. Para iniciar cualquier bloque paralelo es necesaria la sentencia *#pragma omp* donde se incluirá la sentencia adecuada para la tarea a realizarse con sus respectivas cláusulas. En este caso, *parallel*, indicó una sección paralela, y su cláusula *private* declara que variables serán privadas para el bloque.

También tenemos en el ejemplo dos llamadas a funciones de OpenMP de tiempo de ejecución, que proporcionan información al programa sobre su ejecución. Tenemos *omp_get_thread_num()* que proporciona la identidad del hilo que se encuentra ejecutándose. Por el otro lado, tenemos *omp_get_num_threads()* que nos regresa la cantidad de hilos ejecutándose en este bloque paralelo. *omp_set_num_threads(4)* por el otro lado, limita el número de hilos a únicamente 4, en este caso.

4.2. Directiva for

Uno de los casos más recurrentes dentro de las tareas de programación son los ciclos. A través de cientos, miles o millones de iteraciones de ciclos se logra procesar información valiosa en en otros tiempos parecía imposible calcular. Sin embargo, si los cálculos son muy grandes también las computadoras llegan a su límite. Por lo tanto, es bueno repartir esta carga de procesamiento en diferentes hilos. El problema de encontrar números primos es uno de los problemas más antiguos, y no se ha encontrado fórmula alguna para poder encontrarlos a todos mediante una fórmula. La único que es posible es comprobar número por número si es divisible entre algún otro número. Esta tarea es tediosa y larga y en ciertos momentos incomputable. En la siguiente implementación repartiremos el trabajo de un ciclo for entre diversos hilos:

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>

int es_primo(num)
{
    int j, primo;
    for(j=1; j<=num; j++)
    {
        if((num %j)== 0 && num != j && j != 1)
            return 0;
    }
    return 1;
};

int main(int argc, char **argv)
{
    int i, num;
    num = 1000000;
    printf("Son números primos: ");
#pragma omp parallel for
    for(i=1; i<=num; i++)
    {
        if(es_primo(i))
            printf("%d, ", i);
    }
    printf("\n");
    return 0;
}
```

#pragma omp parallel for reparte el trabajo del ciclo. Es posible agregar cláusulas de variables compartidas y de variables privadas, entre otras muchas ya antes descritas. En cada ciclo, se llama a la función *es_primo()*, que regresa 1 si es primo, o 0 si no lo es. En dado caso de que lo sea, se imprime el número en pantalla. Para hacer efectiva la paralelización se utilizó un millón de números, de los cuales se debe encontrar los números primos. El mismo algoritmo secuencial tardó 2 minutos 18.545 segundos, mientras que el paralelo tardó únicamente 31.336 segundos. Este incremento en la velocidad se debió, en parte al no tener variables compartidas que sincronizar.

A continuación veremos un programa sencillo que implementa la sentencia for y además tiene como variable compartida a n y como una variable provada a i:

```
#include<stdio.h>
#include<omp.h>

int main()
```

```
{
    int i, n;
    n=10;
#pragma omp parallel shared(n) private(i)
    {
#pragma omp for
        for(i=0;i<n;i++)
            printf("El hilo %d está ejecutado el ciclo %d\n",
             omp_get_thread_num(), i);
    }
    return 0;
}
```

```
El hilo 4 está ejecutado el ciclo 8
El hilo 4 está ejecutado el ciclo 9
El hilo 2 está ejecutado el ciclo 4
El hilo 2 está ejecutado el ciclo 5
El hilo 1 está ejecutado el ciclo 2
El hilo 1 está ejecutado el ciclo 3
El hilo 3 está ejecutado el ciclo 6
El hilo 3 está ejecutado el ciclo 7
El hilo 0 está ejecutado el ciclo 0
El hilo 0 está ejecutado el ciclo 1
```

En el pasado ejemplo vemos que el grupo de procesadores se reparten el trabajo. Cada hilo comienza a ejecutar un pedazo del ciclo de manera que todos los procesadores están ocupados. La única restricción a la velocidad es la necesidad de compartir una variable, que puede ser accesada de manera múltiple por todos los hilos, lo cual retrasa la ejecución un poco.

```
#include<stdio.h>
#include<omp.h>

int main()
{
    int i, n;
    n=10;
#pragma omp parallel shared(n) private(i)
    {
#pragma omp for
        for(i=0;i<n;i++)
            printf("Region uno: El hilo %d está ejecutado el ciclo
            %d\n", omp_get_thread_num(), i);
#pragma omp for
        for(i=0;i<n;i++)
            printf("Region dos: El hilo %d está ejecutando el ciclo
```

```
            %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

```
Region uno: El hilo 0 está ejecutado el ciclo 1
Region uno: El hilo 2 está ejecutado el ciclo 4
Region uno: El hilo 2 está ejecutado el ciclo 5
Region dos: El hilo 3 está ejecutando el ciclo 6
Region dos: El hilo 3 está ejecutando el ciclo 7
Region dos: El hilo 0 está ejecutando el ciclo 0
```

Como podemos apreciar, al crear dos ciclos *for* dentro de la región *paralell*, cada uno de los bucles ciclos tendrán un orden según se hallan escrito secuencialmente dentro de la región *parallel*. Primero se ejecuta el ciclo 1 y posteriormente el ciclo dos.

4.3. Directiva section

La directiva *section* permite ejecutar una serie de bloques de código de manera paralela, junto a otros bloques. De esta forma, no únicamente se puede repartir las tareas de un ciclo, si no que diversas secciones enteras de código se ejecutarán de manera paralela. Primero se debe llamar a la directiva *parallel*, para después entrar a un bloque de secciones, lo cual se indicará mediante *pragma omp sections*. Cada sección de código separada, a su vez, será llamada desde *pragma omp section*. La llamada será a una función, donde su salida se convertirá a *void*.

```
#include<stdio.h>
#include<omp.h>

void func(int num)
{
    printf("En la funcion no %d que se ejecuta 1\n",num);
    printf("En la funcion no %d que se ejecuta 2\n",num);
    printf("En la funcion no %d que se ejecuta 3\n",num);
    printf("En la funcion no %d que se ejecuta 4\n",num);
}

int main()
{
    int i, n;
    n=10;
#pragma omp parallel shared(n) private(i)
    {
#pragma omp sections
        {
```

```
#pragma omp section
            (void)func(1);
#pragma omp section
            (void)func(2);
#pragma omp section
            (void)func(3);
        }
    }
    return 0;
}
```

```
En la funcion no 3 que se ejecuta 1
En la funcion no 3 que se ejecuta 2
En la funcion no 1 que se ejecuta 1
En la funcion no 1 que se ejecuta 2
En la funcion no 1 que se ejecuta 3
En la funcion no 1 que se ejecuta 4
En la funcion no 3 que se ejecuta 3
En la funcion no 3 que se ejecuta 4
En la funcion no 2 que se ejecuta 1
En la funcion no 2 que se ejecuta 2
En la funcion no 2 que se ejecuta 3
En la funcion no 2 que se ejecuta 4
```

Las instrucciones contenidas en los diferentes secciones se ejecutan cada una en un hilo, por lo que el procesamiento de los tres bloques de código se hace al mismo tiempo.

4.4. Directiva single

Existen ocasiones en que dentro de un bloque paralelo se requiere que un determinado bloque de código sea ejecutado de manera secuencial. Para este propósito tenemos lo cláusula single. Que podemos apreciar con el ejemplo a continuación:

```
#include<stdio.h>
#include<omp.h>

int main()
{
    int n,i,a,b;
#pragma omp parallel shared(a,b) private(i)
    {
#pragma omp single
        {
            a = 10;
            printf("Esto fue ejecutado por el hilo %d\n",
            omp_get_thread_num());
```

```
            printf("Esto fue ejecutado por el hilo %d\n",
            omp_get_thread_num());
            printf("Esto fue ejecutado por el hilo %d\n",
            omp_get_thread_num());
            printf("Esto fue ejecutado por el hilo %d\n",
            omp_get_thread_num());
            printf("Esto fue ejecutado por el hilo %d\n",
            omp_get_thread_num());
        }
#pragma omp for
        for(i=0;i<10;i++)
            printf("Ejecutado %d desde for en el hilo %d\n",
            i, omp_get_thread_num());
    }
    return 0;
}
```

```
Esto fue ejecutado por el hilo 0
Esto fue ejecutado por el hilo 0
Esto fue ejecutado por el hilo 0
Esto fue ejecutado por el hilo 0
Esto fue ejecutado por el hilo 0
Ejecutado 0 desde for en el hilo 0
Ejecutado 1 desde for en el hilo 0
Ejecutado 8 desde for en el hilo 4
Ejecutado 9 desde for en el hilo 4
Ejecutado 6 desde for en el hilo 3
Ejecutado 4 desde for en el hilo 2
```

El código incluido en *single* se ejecuta en un sólo hilo, y de manera secuencial. Esto es útil cuando es necesario tratar una porción de código de manera secuencial.

4.5. Directiva barrier

Barrier es una directiva que nos permite poner un límite a la ejecución de una parte paralela. Al interrumpir la ejecución paralela, se da paso a una nueva zona paralela. Algo parecido pasa con el ciclo *for* o con *sections*. Al terminar el ciclo, OpenMP llama a un *barrier* implícito. Sin embargo, existen ocaciones en que el programador desea controlar manualmente este procedimiento, por lo que existe la posibilidad de llamar a esta directiva.

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>


void print_time(int TID, char *text)
```

```
{
    printf("Hilo %d %s de la barrera en ");
}

int main()
{
    int TID;
#pragma omp parallel private(TID)
    {
        TID = omp_get_thread_num();
        if(TID < omp_get_thread_num()/2) system("sleep 3");
        (void) print_time(TID,"antes");

#pragma omp barrier
        (void) print_time(TID,"despues");
    }
    return 0;
}
```

```
...
Hilo 6 antes de la barrera
Hilo 1 antes de la barrera
Hilo 0 antes de la barrera
Hilo 3 despues de la barrera
Hilo 1 despues de la barrera
Hilo 7 despues de la barrera
...
```

Vemos como existe una paralelización antes de la barrera que no se mezcla con el segundo bloque de instrucciones que son paralelizadas después de la barrera.

4.6. Directiva critical

Cuando existen datos que se comparten entre diferentes hilos existe la posibilidad de entrar en problemas de competencia por los datos. Para evitar que se pierdan datos o se hagan errores por la competencia es posible indicar que la región debe ser ejecutada únicamente por un hilo a la vez cuando entra en cuestión. El siguiente ejemplo muestra la suma de diferentes hilos que se realiza de manera efectiva, sin perder información.

```
#include<stdio.h>
#include<omp.h>

#define TAM 10000

int main(int argc, char **argv)
{
```

```
    int sum = 0;
    int n,TID,sumLocal,i;
    int a[TAM];
    n = TAM;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
    {
        TID = omp_get_thread_num();
        sumLocal = 0;
#pragma omp for
        for(i=0; i<n; i++)
            sumLocal += a[i];
#pragma omp critical(update_sum)
        {
            sum += sumLocal;
            printf("TID=%d: sumLocal=%d sum=%d\n",TID,sumLocal,sum);
        }
    }//Fin de la regi n paralela
    printf("Valor de sum después de la regi n paralela: %d", sum);
    return 0;
}
```

```
TID=1: sumLocal=74980000 sum=74980000
TID=0: sumLocal=24980000 sum=99960000
TID=5: sumLocal=274980000 sum=374940000
TID=4: sumLocal=224980000 sum=599920000
TID=2: sumLocal=124980000 sum=724900000
TID=6: sumLocal=324980000 sum=1049880000
TID=3: sumLocal=174980000 sum=1224860000
TID=7: sumLocal=374980000 sum=1599840000
Valor de sum después de la regi n paralela: 1599840000
```

4.7. Directiva atomic

En algunas máquinas se incluye optimización para manejar las competencias. Únicamente la parte izquierda de la operación de asignación se protege de competencias, no la parte derecha. Y únicamente se permite una operación de asignación con la directiva. Si no se encuentra una aceleración hardware para atomic, se utiliza como una sección critical.

```
#include<stdio.h>
#include<omp.h>

#define TAM 10000

int main(int argc, char **argv)
{
```

```
    int n,i;
    int ic = 0;
    n = TAM;
#pragma omp parallel shared(n,ic) private(i)
    for(i=0; i<n; i++)
    {
#pragma omp atomic
        ic += 1;
    }//Fin de la regi n paralela
    printf("Contador: %d\n", ic);
    return 0;
}
```

```
Contador: 80000
```

5. Implementación

5.1. Implementación de búsqueda secuencial

Ahora mostraremos el mejoramiento de la velocidad al tratar datos grandes mediante un programa sencillo de búsqueda secuencial. En este caso utilizamos la paralelización con la cláusula *for* de *parallel*, y lo comparamos para cada número de datos en secuencial. Primero mostramos el código fuente de la búsqueda secuencial.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#include "omp.h"


#define MAX_SIZE_VISIBLE 100

int main(int argc, char **argv)
{
    int *a;
    unsigned long long int size, i;
    double start, end;
    double parallel, sequential;
    int n, j;

    // Es necesario tener dos argumentos en la linea de comandos
    // de lo contrario se interrumpira el programa y mostrará
    // un mensaje explicando el unos del mismo.
```

```
    if(argc <= 2)
    {
        printf("Faltan argumentos, el uso correcto es:
        \n\tseq [tamano de lista] [numero a buscar]\n");
        return 1;
    }

    // Transformamos los argumentos en números
    size = strtoull(argv[1], NULL, 10);
    n = atoi(argv[2]);

    printf("Elementos en la lista: %llu \nNumero a
    buscar: %d\n", size, n);

    // Alojando dinamicamente un arreglo para guardar los valores
    // aleatorios.
    a = malloc(size * sizeof(unsigned long long int));

    // Generacion de numeros aleatorios.
    printf("Generando numeros aleatorios en la lista...\n");
    srand(time(NULL));

    for(i=0;i<size;i++)
    {
        a[i]=rand()/1000;
    }


    if(size<MAX_SIZE_VISIBLE)
    {
        for(i=0;i<size;i++)
        {
            printf("%llu: %d\n",i,a[i]);
        }
    }
    else
    {
        printf("Lista demasiado grande para mostrarse
        en pantalla.\n");
    }

    // Inicio de busqueda secuencial
    printf("Buscando...\n");
```

```
    j = 0;
    start = omp_get_wtime();
    for(i=0;i<size;i++)
    {
        if( a[i]==n )
        {
            // Impresion de resultados
            printf("%d se encuentra en la posicion
            %lu\n",n,i);
            j = 1;
        }
    }
    if(!j)
    {
        printf("No se encontro el numero\n");
    }
    end = omp_get_wtime();
    printf("Procesamiento secuencial %f\n", end-start);
    sequential = end-start;


    j = 0;
    start = omp_get_wtime();
    #pragma omp parallel for default(none)
    firstprivate(size, n, a) private(i) shared(j)
    for(i=0;i<size;i++)
    {
        if( a[i]==n )
        {
            // Impresion de resultados
            printf("%d se encuentra en la posicion
            %lu\n",n,i);
            j = 1;
        }
    }
    if(!j)
    {
        printf("No se encontro el numero\n");
    }
    end = omp_get_wtime();
    printf("Procesamiento paralelo %f\n", end-start);
    parallel = end-start;

    printf("\n\n          Tamaño\tSecuencial
    \tParalelo\n");
```

```
    printf("Resultado: %lu \t %f \t %f \n", size, sequential,
    parallel);


    // Liberando la memoria recervada
    free(a);

    return 0;
}
```

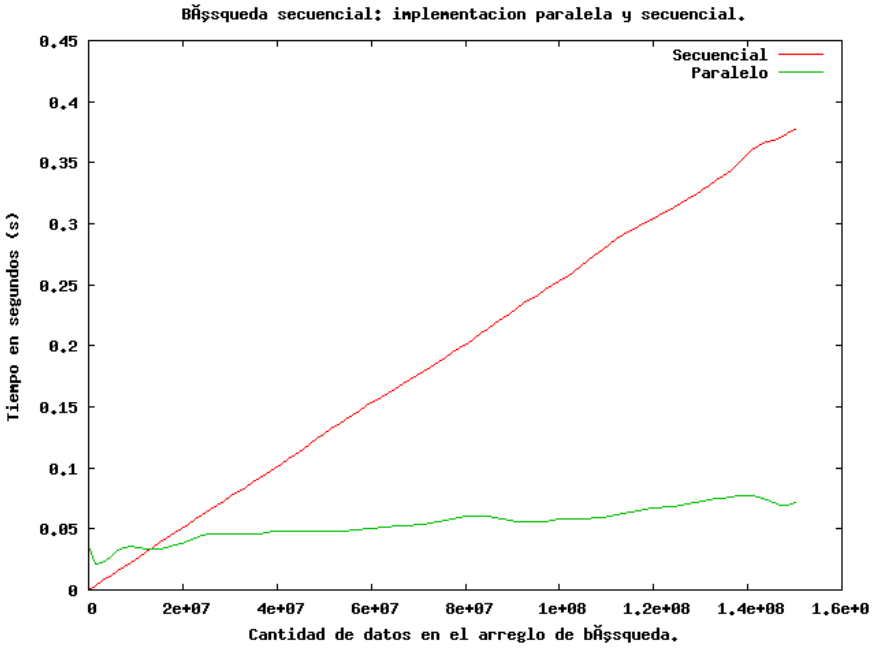
Para graficar creamos un script *graficar.sh* que utiliza la herramienta GNUPlot.

```
#!/bin/bash

# Programa creado para evauar el rendimiento
# de la versi n paralela y secuencial del
# algoritmo de búsqueda secuencial.

echo " Bienvenido a la prueba de rendimeinto de búsqueda secuencial."

echo " + Generando lista de pruebas..."
i=0
vals[0]=1

while [ $i -lt 150 ]
do
    valtmp=${vals[$(( $i ))]}
    newval=$(echo "$valtmp+1000000" | bc)
    i=$(( $i + 1 ))
    vals[$i]=$newval
done

echo " + Iniciando el cíclo de búsquedas..."
echo "" > data.dat
for val in "${vals[@]}"
do
    echo " - Buscando en un arreglo de tamaño: ${val}"
    ./seq_par "${val}" 5342 | awk $info '/^Resultado/ { print $2 " " $3
        " " $4 }' >> data.dat
done
echo " + Graficando los datos recopilados"
gnuplot < plot.gpi
cat data.dat
echo " + Finalizado! $2"
```

Complementado de las intrucciones de GNUPlot con el archivo *plot.gpi*:

```
set terminal png
set output "bus_seq.png"
set title "Búsqueda secuencial: implementacion paralela y secuencial."
set xlabel "Cantidad de datos en el arreglo de búsqueda."
set ylabel "Tiempo en segundos (s)"
plot "data.dat" using 1:2 title "Secuencial" with lines smooth bezier,
"data.dat" using 1:3 title "Paralelo" with linespoints smooth bezier
```

Veamos ahora el resultado:



Como podemos observar, el algoritmo de búsqueda secuencial es más afectivo ejecutado de una manera secuencial sobre un conjunto de datos reducidos, ya que la OpenMP requiere una cierta cantidad de tiempo para iniciar la ejecución como librería y para iniciar su tiempo de ejecución. Sin embargo, la versión en paralelo del algoritmo es mucho más eficiente con datos a partir de la medición realizada con un arreglo de 4000001 sobre el cual se realiza la búsqueda.

Existe también una importante variación en el tiempo de ejecución del la versión en paralelo debido a que el proceso depende de otros factores que afectan el tiempo de ejecución, como lo es el uso de los procesadores por otras aplicaciones. Sin embargo, en lo general se puede ver una tendencia a mantener un ligero incremento en el tiempo de búsqueda en comparación con la fuerte tendencia positiva del tiempo requerido por la versión secuencial.

La principal limitante que se encontró al ejecutar el programa, no fue el procesamiento, ya que los 16 cores de la máquina sobre la cual se corrió permitieron mejorar de manera importante el rendimiento, si no que fue la memoria disponible para poder almacenar el arreglo que generamos. Podrían crearse alternativas para tratar este tema, como lo es paginación, sin embargo, al momento de escribir aplicaciones que trabajen con grandes cantidades de datos es importante tener en cuenta la memoria de la que se dispone.

Como pudimos ver, OpenMP y la programación en paralelo en general nos aportan importantes herramientas para poder escribir programas que utilicen al máximo las nuevas tecnologías multi core. Si queremos escribir programas eficientes hoy en día es necesario utilizar una programación paralela.

Referencias

[1] Chapman, Barbara. et all. Using OpenMP, Portable Shared Memory parallel programming. ed. The MIT Press(England: London, 2008).

[2] Pacheco, Peter. A Introduction to Parallel Programming. ed. Morgan Kaufman (Estados Unidos: Burlington, 2011).

[3] Matloff, Norm. Programmin on Paralell Machines. ed. University of California. (Online: http://heather.cs.ucdavis.edu/matloff/158/PLN/ParProcBook.pdf)

[4] Official OpenMP Specifications, Summary Card C/C++ http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf

[5] http://en.wikipedia.org/wiki/OpenMP (Consultado el 10 de Noviembre de 2012)