Revista de Investigación de Física **14**, 111401851 (2011)

Revisitando la programación en paralelo

A. P. Aslla[1,2], R. A. Montalvo[1] y P. H. Rivera[1]

[1] *Facultad de Ciencias Físicas, Universidad Nacional Mayor de San Marcos, Lima 1, Perú*
[2] *Universidad Nacional Tecnológica del Cono Sur de Lima, Lima, Perú*

Recibido 15 noviembre 2010 – Aceptado 27 diciembre 2011

En la última década ha hecho su aparición los procesadores de varios núcleos, de modo que la programación en paralelismo multisimétrico (SMP) ha vuelto a la vida luego de un periodo de una década en que los *clusters* basados en el OpenMPI habian casi dominado el mundo del procesamiento de alto desempeño. Actualmente tenemos en el mercado los procesadores con 2, 4 y 6 núcleos; los procesadores Cell Be con un núcleo maestro, *Power Processor Element* (PPE) y ocho núcleos denominados *Synergistic Processor Element* (SPE), y los procesadores gráficos que poseen desde 48 hasta 448 núcleos dedicados al cálculo numérico.
En este trabajo mostramos nuestra experiencia de la interacción con estos procesadores, discutiendo las técnicas de programación y las posibilidades futuras para la realización de cálculo numérico de bajo costo.
Palabras claves: OpenMPI, OpenMP, Cell BE, i7 980X, CUDA.

Revisiting the parallel programming

In the last decade the multicore processors had emerged with the comeback of the symmetric multiprocessing (SMP) after a long decade in which the based OpenMPI clusters had dominated the high performace processing world. Actually, we have in the market 2, 4 and 6 multicore processors; the Cell BE processors with a Power Processor Element (PPE) and 8 vectorial processors called Synergistic Processor Element (SPE), and graphics processors which have from 48 to 448 cores dedicated to numerical tasks.
In the present work, we show our experience in the interaction with these processors, discussing the programming techniques and the future possibilities for making low cost numerical tasks.
Keywords: OpenMPI, OpenMP, Cell BE, i7 980X, CUDA.

La física computacional y las ciencias de la computación se han desarrollado de manera paralela retroalimentándose mutuamente. Desde la aparición del UNIX en 1969 [1], estos esfuerzos han sido cada vez más fructíferos en el establecimiento de Internet en 1979 con el modelo cliente-servidor que ha permitido el crecimiento exponencial de la Internet alrededor del mundo.

El propio UNIX fue el primer esfuerzo por hacer portable un sistema operativo a otras máquinas de diferentes empresas fabricantes, pues el UNIX que fue codificado inicialmente en assembler para una máquina PDP-11/20, fue codificada en 1972 usando el lenguaje C para poder ser usada en otras máquinas solamente cambiando un pequeño *kernel* con las instrucciones assembler correspondientes a otras máquinas.

Este simple hecho de su portabilidad a otras máquinas y el hecho de que la ATT, propietaria de los Laboratorios Bell, estuviera prohibida por ley de comerciar productos que no tuvieran que ver con la telefonía, permitió que el UNIX se distribuyera de manera abierta en las universidades incluyendo las fuentes de su código, tanto en C como el kernel desarrolado en assembler, del PDP-11720, lo que indudablemente, estimuló el alto desarrollo del UNIX que hasta el día de hoy observamos su influencia en el Linux, en los sistemas Apple TV, iOS y Mac OS X de la Apple y en todos los dispositivos móbiles que usan el Android.

Desde los años 80's, para realizar cálculos de gran envergadura surgió el interés de varias empresas de desarrollar la supercomputación. Para ello se adoptaron diferentes estrategias, una de ellas consistía en construír placas madres con varios procesadores compartiendo un solo banco de memorias, para el control de dicha estrategia se desarrolló el *Symmetric Multi-Proccessing*,

SMP, que consistía en un conjunto de instrucciones que controlaban el tráfico entre los diferentes procesadores y el acceso a la memoria.

En los años 90's surgió otra estrategia que consistía en establecer un código de comunicación común entre diferentes máquinas cuyas placas madres eran uniprocesadores con su respectiva memoria. Este conjunto de instrucciones se denominó *Message Passing Interface*, MPI. Colocando estas instrucciones en cada máquina de cualquier fabricante que usara cualquier variante de UNIX o Linux, estas conseguían comunicarse entre sí compartiendo los procesadores y las memorias particulares de cada equipo bajo la filosofía del servidor maestro y los servidores esclavos. En los últimos años se establece un nuevo estándard y portable denominado OpenMPI [2].

En los últimos años, con el surgimiento de los procesadores de varios núcleos, nuevamente la filosofía SMP comienza a resurgir y se establece un nuevo estándard portable y abierto denominado OpenMP [3].

En la primera década del presente siglo han surgido otras estategias que están basados en el diseño del procesador. El primero es el proyecto de la SONY, TOSHIBA e IBM que se desarrolla entre el 2001-2004 desarrollando el procesador *Cell Broadband Engine Architecture*, simplemente Cell BE [4], para tareas de cálculo de alto desempeño. El segundo es el proyecto *Compute Unified Device Architecture*, CUDA [5], de la empresa Nvidia que produce un ambiente de programación C usando sus placas gráficas para cálculo de alto desempeño.

En las próximas secciones discutimos los detalles de estas nuevas posibilidades de usar la programación en paralelo usando recursos computacionales de bajo costo.

OpenMPI

OpenMPI es el acrónimo de *Open Message Passing Interface* [2], que consiste de un conjunto de instrucciones que conforman una librería que permite la comunicación entre los procesadores y las memorias dispersas en una red en código abierto. En el OpenMPI se han unido los mejores desarrollos sobre el *MPI* que se han realizado en los últimos quince años, estos son los desarrollos del FT-MPI de la Universidad de Tennesse, LA-MPI del Laboratorio de Los Álamos y del LAM-MPI de la Universidad de Indiana. Nuestro grupo ha tenido dos experiencias previas de configurar dos *cluster*s basado en el LAM-MPI de la Universidad de Indiana en los últimos diez años.

Las diez primeras de las TOP500 supercomputadoras [6] del mundo basan su actividad computacional en esta librería[1]. Puesto que estas máquinas constan de varias centenas de placas madres cada una con un conjunto de procesadores de modo que la comunicación entre las placas e inclusive los mismos procesadores de la misma placa está basada en el OpenMPI, esta comunicación tiene que ser lo suficientemente rápida en vista de que la latencia de los procesadores disminuya, definiendo la latencia como la diferencia entre el tiempo de espera de una instrucción y la siguiente, y el tiempo de ejecución de la instrucción, siendo el primero mucho mayor que el segundo. Para ello se necesita de enlaces de redes que permita la comunicación en decenas o centenas de GBytes por segundo. Eso se obtiene usando fibras ópticas de gran desempeño.

Para las supercomputadoras y para pequeños *cluster*s que están en un espacio se les denomina sistema paralelo no distribuído, pero si las computadoras que conforman el sistema se encuentran en diferentes lugares de un edificio o incluso dentro de una universidad se les denomina sistema paralelo distribuído. En ambos casos, siempre están interconectados por ruteadores. También se les denomina sistema dedicados y no dedicados, respectivamente.

El OpenMPI trabaja con el concepto de servidores *maestro-esclavos*, en el cual debe escogerse la máquina que debe ser el *maestro* y los demás deben ser los esclavos. El *maestro* es la máquina al cual se tendrá acceso desde internet a través de un *firewall*. El *maestro*, en un sistema no distribuído, tendrá acceso a los *esclavos* mediante una red privada a través de una segunda placa de red. En un sistema distribuído, la comunicación entre el *maestro* y los *esclavos* es a través de la red de internet, lo que permite una comunicación más lenta entre los nodos debido al intenso tráfico de la red local y una mayor latencia de los procesadores.

La primera etapa para el proceso de programación en un conjunto de máquinas, distribuídas o no, usando el paralelismo es analizar si el código fuente del programa tiene segmentos que pueden ser paralelizados, porque si todos los segmentos son en serie de modo que los cálculos de los datos dependen de los resultados anteriores (métodos autoconsistentes), en este caso no hay forma de paralelizar. Lo que se sugiere aqui es usar el procesador más rápido del mercado sin pretender usar tampoco todos los núcleos mediante el OpenMP.

Si uno encuentra un `do` inicial que es paralizable y que abarca casi todo lo que se quiere calcular, esto permite dividir el trabajo de cálculo entre el número total de máquinas que se tengan en el sistema en paralelo con las instrucciones siguientes:

```
 Program test
 include "mpi.h"
! aquí se define los vectores y matrices
```

[1] El total de núcleos que usan estas máquinas está en torno de los 750,000.

```
 ...
 ..
 .
! aquí se inicializa el proceso en paralelo
 call MPI_INIT(ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD,idproc,ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
! se definen los datos del programa
 ...
 ..
 .
! aquí se corre el proceso en paralelo
 call MPI_BCAST(ndat,1,MPI_INTEGER,0,
+        MPI_COMM_WORLD,ierr)
 do inf=idproc+1,ndat,numprocs
 ...
 ..
 .
 end do
! aquí se termina el proceso en paralelo
 call MPI_FINALIZE(irc)
 ...
 ..
 .
 end Program test
```

La primera línea del código es la instrucción Fortran del inicio del programa. La segunda línea es `include "mpi.h"`, esta instrucción es muy importante ya que es la que informa al compilador el camino, `/usr/include`, donde buscar el archivo `mpi.h`, en la que se encuentran declaradas las variables que usa el MPI. A continuación viene las declaraciones de las constantes, variables, vectores y matrices que se usan en el programa de cálculo.

Luego, viene en forma consecutiva tres instrucciones que llaman a las subrutinas que se encuentra en la librería MPI para realizar las tareas específicas correspondientes a cada instrucción. La primera instrucción, `call MPI_INIT(ierr)`, es una llamada a la subrutina `MPI_INIT` que inicializa el sistema en paralelo y el sistema responde mediante la variable entera `ierr` de modo que si es 0, el sistema está en la espera de más instrucciones.

La segunda es una llamada a la subrutina `MPI_COMM_RANK` que identifica las máquinas disponibles en el sistema en paralelo. Esta subrutina responde con las variables `MPI_COMM_WORLD` que es el comunicador con todos los nodos disponibles, `idproc` es la respuesta de cada nodo identificándose a sí misma con un número entero a partir de 0 y la variable entera `ierr` resulta 0 si la identificación de los nodos a sido correcta.

La tercera es una llamada a la subrutina `MPI_COMM_SIZE` que identifica el número de maquinas disponibles en el sistema en paralelo. La respuesta de esta subrutina usa las variables `MPI_COMM_WORLD`, `numprocs` y la variable entera `ierr`. La primera y la última respuesta es similar al caso anterior y la seguna nos proporciona el número total de procesadores.

La instrucción que comienza la corrida en paralelo y el proceso de dividir las tareas equitativamente para cada procesador está dado por la llamada a la subrutina `MPI_BCAST(ndat,1,MPI_INTEGER,0,MPI_COMM_WORLD, ierr)`, donde `ndat` es cantidad de iteraciones, `1` es la identificación por default del master preguntando al nodo si es el `1` y el nodo responde proporcionando su identificación vía `MPI_INTEGER`, el `0` es la identificación por default del master al preguntar por el número total de nodos y el sistema responde con el comunicador `MPI_COMM_WORLD` que abarca a todos los nodos y si todo sale bien el valor de `ierr` es igual a 0. Aquí hay que considerar que `ndat` sea un múltiplo entero del número de nodos de modo que la carga de procesamiento sea igual para cada nodo, siempre que todos los nodos posean las mismas características de *hardware*.

A partir de aquí, la instrucción Fortran `do inf=idproc+1,ndat,numprocs`, hace la división adecuada para cada nodo donde los datos del `do` van variando desde 1 que corresponde al nodo 0, 2 al nodo 1, 3 al nodo 2 y así sucesivamente hasta alcanzar el primer bloque con el último nodo. Luego que terminan la tarea, recomienza la segunda asignación y así sucesivamente. Usualmente se observa que un nodo termina su asignación antes que los demás y para no esperar a los demás el toma su asignación que le corresponde saltando el valor `numprocs` que corresponde al número total de nodos y así realiza una ganancia de tiempo. Aquí cabe resaltar que aún teniendo todas las máquinas con las mismas especificaciones técnicas, existe algunas que son más rápidas que las otras.

La instrucción que termina el proceso en paralelo es `MPI_FINALIZE(irc)` que responde con el valor entero 0 para la variable entera `irc` si el proceso termina con éxito. Otros valores diferentes de 0 indican diferentes errores de ejecución que deben ser analizados con el manual del OpenMPI [2].

Si uno posee un conjunto de máquinas de diferentes arquitecturas y distintas generaciones con desempeños bastante diferenciados unos de otros, el OpenMPI ofrece dos instrucciones, la primera que envia la tarea específica y la segunda trae los resultados luego de realizado los cálculos. Estas instrucciones deben ser cuidadosamente calibradas para dar la carga adecuada a cada nodo.

```
...
integer status(MPI_STATUS_SIZE)
...
..
.
```

```
 call MPI_SEND(field,ndat,MPI_DOUBLE_PRECISION,
+         idproc,1,MPI_COMM_WORLD,ierr)
 ...
 ..
 .
 call MPI_RECV(field,ndat,MPI_DOUBLE_PRECISION,
+         idproc,1,MPI_COMM_WORLD,status,ierr)
```

La primera entrada en las subrutinas es la variable que se desea ingresar y la que debe salir. Es recomendable colocar la misma variable para ahorrar memoria cuando se use el OpenMP en un `do` interno, cuyo uso se explica en la próxima sección. En nuestro caso usamos `field`. La siguiente entrada es un número entero relacionado con la cantidad de datos máximos que se ejecuta. La tercera explica el tipo de variable, la cuarta es la identificación del nodo al que va a ser enviado la tarea, la siguiente es el membrete del nodo. Las otras entradas y salidas son similares a los casos anteriores, excepto `status` que es un variable entera que debe declararse al inicio y realiza una llamada a la subrutina `MPI_STATUS_SIZE` significando que la tarea encomendada ha sido realizada.

La librería del OpenMPI ofrecen más de 200 instrucciones que están diseñadas para realizar una programación personalizada para los procesadores y las memorias de manera individual. Esto depende del tipo de solución numérica que se ha escogido. Los detalles de los mismos están en los manuales del OpenMPI [2].

Finalmente, en la máquina que se usa como *maestro* se compila el código usando la instrucción `mpif95 -o file. file.f90 -lmpi`, luego se activa el programa ejecutable con `mpirun -np 10 file &`, donde la opción dada por `-np 10` indica el número de nodos a ser usado, es decir, que tenemos una red con 10 nodos, un *maestro* y nueve *esclavos*, que están atentos a nuestra llamada del OpenMPI para correr el proceso en paralelo. Esto significa que en las 10 máquinas deben estar instaladas el OpenMPI, el mismo usuario debe tener acceso a las 10 máquinas, es decir se debe tener un área de usuario en las 10 máquinas y usando el `openssh`, se debe tener acceso a través de este `shell` a las 10 máquinas. El programa compilado debe ser copiado en todas las máquinas, en el mismo directorio que en la máquina *maestro*. Una alternativa a esta forma de acceso directo a las máquinas es el *Network File System*, NFS [7], que todas las distribuciones Linux lo traen preinstalados. El NFS es una sistema de archivos en red en la que el directorio del usuario, *home*, está en el maestro y es exportada virtualmente a todas las máquinas esclavas, de modo que, el usuario accede a su directorio personal en cualquier máquina esclava usando su nombre de usuario y contraseña que ha sido incorporada al registrar su apertura en la máquina maestro. Sólo hay que activarlo y usar el *yellow pages* o *Network Information System*, NIS [8], para compartir los nombres de usuarios y las contraseñas en todas las máquinas clientes. Esta es una alternativa cuando se usan centenas de máquinas en la paralelización, el único inconveniente es que se incrementa el tráfico en la red que puede perturbar las comunicaciones del OpenMPI. Existe una nueva versión del NFS, llamada de *Parallel Network File System*, pNFS [9], que permite un acceso directo a los dispositivos de depósitos de datos, que ciertamente era el gran problema usando el MPI y que congestionaba la red y provocaba mucha latencia en los procesadores.

OpenMP

El OpenMP es el acrónimo de *Open Multi-Proccessing*, multiprocesamiento simétrico de código abierto. Es un conjunto de instrucciones de programación que han sido adheridos a los *kernels* de los sistemas operativos y compiladores para controlar el flujo de datos e instrucciones entre los diferentes núcleos y sus correspondientes áreas de memoria. Esta técnica de programación comienza en los años 80 cuando surge la necesidad de colocar varios procesadores en una placa madre de modo que comparten una solo conjunto de bancos de memoria. La técnica se llamó simplemente de *Symmetric Multi-Proccessing*, SMP, y aquí venía el problema, cada empresa que fabricaba su placa madre con 2, 4, 8, ..., 2^N procesadores, donde N es el número de procesadores, tenía su propio código de programación. Las empresas que desarrollaron más esta codificación fueron la *Thinkin Machines* [10], *Cray* [11], *nCUBE* [12], *Kendall Square Research*, *Intel Supercomputing Systems Division*, *MasPar* y *Meiko Scientific*. No existía compatibilidad ni portabilidad entre la codificación de los programas que se diseñaban en esas maquinas. Eran exclusivas de cada empresa.

Con el surgimiento de los procesadores de varios núcleos y su relativo bajo precio comparado con las máquinas de empresas mecionadas líneas arriba, surge la necesidad de establecer un código de programación abierto y universal que las diferentes máquinas puedan entender y más aún que deben ser incorporadas en los diferentes compiladores para que los códigos que se elaboren sean portables y compatibles con las diferentes versiones. Este proceso de estandarización se inicio a fines de los 90's y en alcanzó su madurez en los años 2005-2008, y se denomina OpenMP que es una versión de código abierto del SMP.

Para la paralelización basado en el OpenMP depen-

[2]Las memorias denominadas L1, L2 y L3, indican el nivel (*level*), L1 indica que está en el procesador, son pequeñas y tan rápidas como el procesador; L2 indica una memoria próxima al núcleo de mayor capacidad que L1, pero de mayor latencia o velocidad media. Y L3 indica la memoria más alejada de mayor capacidad pero de menor velocidad.

de estrictamente de la cantidad de memoria cache $L3^2$ que posee cada núcleo. Los procesadores Intel i3, i5 e i7 poseen de 2 a 6 seis núcleos físicos. Por cada núcleo físico según el sistema operativo compatible con la tecnología *Hyper-Threading* [13] emula 2 núcleos virtuales denominados *threads*. Por ejemplo, el procesador i5-480M de 2.67 MHz posee dos núcleos fijos y mediante la tecnología *Hyper-Threading* de la Intel direcciona 4 núcleos virtuales todos ellos compartiendo una memoria caché de 3 MB, esto significa que cada núcleo tiene acceso a 0.76 MB para depositar sus datos e instrucciones. Mientras que el procesador i7-2670QM de 2.20 GHz con memoria caché de 6.0 MB tiene 4 núcleos físicos y 8 *threads* o núcleos virtuales cada uno accediendo a 0.75 MB de la memoria caché. El procesador i7-980X de 3.3 MHz posee seis núcleos físicos y mediante la tecnología *Hyper-Threading* emula 12 núcleos virtuales con un memoria caché de 12 MB de modo que cada núcleo accede a 1.0 MB de memoria. Esto significa que, si queremos invertir matrices de mayor tamaño debemos usar el procesador que tenga mayor tamaño de memoria caché para que la paralelización sea efectiva con el consiguiente ahorro de tiempo en las corridas de las simulaciones.

Mostramos a continuación la estrategia para codificar un buen programa que utilize la potencia del OpenMP. En primer lugar, se identifica el `do` más interno que realiza los cálculos con mayor intensidad y donde el número de vectores y/o matrices sean menores. Luego se añaden las instrucciones del OpenMP para la paralelización dentro de ese `do` con la condición de que $N_d = N_t \times N_g$, donde N_d es el número de datos, N_t el número total de núcleos y N_g es el número entero de grupos de datos o corridas y depende de N_d. Para una figura tridimensional con buena resolución para ser publicada en una revista de doble columna es suficiente que $N_d \approx 500$ por dimensión. Si tenemos 12 núcleos el número de corridas para alcanzar un valor próximo a 500 es 42, de modo que $42 \times 12 = 504$. Para el fortran 95 las instrucciones OpenMP más elementales se muestran a continuación

```
  do inf=1,ndat   ! 1--> ndat Barrido para campo
                  ! magnetico
     xn=xn0+(xp/xq)*(inf-1)/float(ndat-1)
     call cpu_time(tnow)
     print*,inf,xn,(tnow-tini)/60,'min'
     ! Comienza la paralizacion en openmp
     ! Se calcula los elementos de las matrices
     ! del sistema dependen del N° de fotones y
     !del barrido de la energia
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP SHARED(qf,xn,cx,fx,cy,fy,v1,v2,e0,hw, &
!$OMP ener2,ener1,epsilon,jlx,lxy,ccx,idth) &
!$OMP PRIVATE(ien,qen,ifot,qea,qeb,qec,qda, &
!$OMP green,qtraza,staten)
!$OMP DO
     do ien=1,ndat ! comienza el barrido de
                   ! energia
     ...
     ..
     .
     end do  !termina el barrido de energia
!$OMP END DO
!$OMP END PARALLEL
  end do !termina el barrido de campo magnetico
```

Las instrucciones OpenMP comienzan siempre desde el primer espacio de la línea con el símbolo `!\$OMP`. La instrucción `PARALLEL DEFAULT(NONE)` es la primera instrucción que indica al compilador que debe prepararse para recibir instrucciones OpenMP. La instrucción `SHARED`(...) indica las variables y constantes que deben ser compartidas por todo los núcleos. Estas pueden ser cantidades escalares, vectores o matrices; pueden ser caracteres, enteras, reales o complejas. La instrucción `PRIVATE(...)` indica las variables o constantes cuyos valores son definidos y usados por cada núcleo de manera privada. Los valores de estas constantes o variables son únicas para cada núcleo y son definidas en la instrucción `!$OMP DO` que inicia la paralelización. El siguiente `do` como en nuestro ejemplo `do ien=1,ndat ! comienza el barrido de energia`, donde `ndat` es igual al número de núcleos multiplicado por el número de corridas, inicia el detalle del proceso en paralelo. Este proceso termina con un `end do` y a continuación las instrucciones `!$OMP END DO` y `!$OMP END PARALLEL` que indican al compilador que el proceso en paralelo SMP ha finalizado.

Para la compilación usando el ifort de la Intel se usa el comando `ifort -openmp -O -o file file.f90` donde `file` es el nombre del archivo del código fuente del programa. Para correr el código compilado simplemente usar el nombre de archivo `file` o según la configuración de su sistema `./file`.

Cell BE

Desde los años 70's se han desarrollado dos tipos de procesadores que son los denominados *Complex Instruction Set Computing*, CISC y los *Reduced Instruction Set Computing*, RISC. Los segundos se han caracterizado por tener un mejor desempeño que los primeros. Entre los segundos se pueden considerar los procesadores DEC Alpha, AMD 29k, ARC, ARM, Atmel AVR, Blackfin, MIPS, PA-RISC, Power, PowerPC, SuperH y SPARC [14] que permitió el desarrollo del mercado de las *workstations*, estaciones de trabajo, en los años 90's, pero ahora su uso es más intensivo en los autos, en los

celulares, televisores y toda la gama de productos electrodomésticos.

Para el año 2000, los procesadores CISC que son básicamente los producidos por la Intel y los procesadores RISC casi no se diferenciaban en su desempeño. Aunque el predominio de la Intel en el mercado de las computadoras domésticas le han permitido mantener toda una línea de investigación y desarrollo para optimizar los procesadores CISC con la tecnología del *Hyper-threading* y la construcción de procesadores con varios núcleos. Pero esta idea revolucionaria, viene de un proyecto de la SONY, TOSHIBA e IBM que entre el 2001 y el 2004 desarrollan el procesador *Cell Broadband Engine Architecture* o simplemente Cell BE [4], continuando la filosofía RISC.

El Cell BE está formado por un procesador de 64 bits con la arquitectura Power denominado *Power Processor Element*, PPE, con memoria L1 de 32 kB y memoria L2 de 512 kB y 8 procesadores vectoriales independientes de 128 bits denominados *Synergistic Processor Elements*, SPE, con memoria LS de 256 kB cada uno. En términos más simples, el procesador PPE es el distribuidor de las tareas específicas de cálculo para cada procesador SPE. Cada SPE solo se dedica a realizar los cálculos que le son encomendados. Esto implica hacer dos programas, uno para el PPE y otro para los demás SPE. Si cada SPE se dedica a ser un tipo de cálculo, entonces se debe crear un programa de cálculo para cada SPE individual. Pero si los cálculos son iguales y solo varía los rangos de datos que cada SPE debe realizar, entonces en el programa del PPE se debe establecer que rangos van para cada SPE. Si comparamos la técnica de programación del Cell BE con las del OpenMPI y OpenMP, se debe afirmar que las filosofías de programación son totalmente diferentes y no existe forma de concluir cuál es la de mayor performance. Para los procesadores Cell BE, la IBM ha desarrollado el XL Fortran que compila los programas tanto para el PPE como para los SPE, así como el XL C/C++ para los códigos en C y C++ [15].

Las máquinas *Play Station 3* de la SONY poseen un Cell BE que al ser instalado el *Yellow Dog Linux*, puede ser utilizado todo el potencial que ofrece este procesador con los compiladores XL Fortran y XL C/C++ desarrolladas por la IBM. Esta es la forma más económica de aprender estas herramientas de alta performance sin tener que adquirir las máquinas IBM con la tecnología *Power*. La desventaja de usar el PS3, es que dos de los procesadores SPE están desactivados, de modo que la respuesta de un PS3 presenta un mensaje como

```
pablo@silmaril[301]:spu-gcc43 simple_spu.c -o
simple_spu
pablo@silmaril[302]:ppu-embedspu simple_spu
simple_spu simple_spu_csf.o
pablo@silmaril[303]:gcc simple.c
simple_spu_csf.o -lspe2 -o simple
pablo@silmaril[304]:./simple
Hola San Marcos desde el Cell-SPU (0x10018008)
Hola San Marcos desde el Cell-SPU (0x10018688)
Hola San Marcos desde el Cell-SPU (0x10018920)
Hola San Marcos desde el Cell-SPU (0x10018b98)
Hola San Marcos desde el Cell-SPU (0x10018e10)
Hola San Marcos desde el Cell-SPU (0x10019088)

El programa se ha ejecutado sin problemas.
pablo@silmaril[305]:
```

En el mensaje anterior se aprecia, en la primera línea, la compilación del programa para los SPU con el compilador `spu-gcc43` del programa `simple_spu`. Luego el compilador `ppu-embedspu` encapsula el código para los SPE. La tercera línea se usa el `gcc` para compilar el programa para el procesador PPE, `simple.c`, y enlaza a los binarios producidos con los dos comandos anteriores.

Un ejemplo de dos pequeños códigos para los procesadores PPE y SPE del Cell BE, respectivamente, son mostrados a continuación.

```
#include <stdio.h>
#include <libspe.h>
extern spe_program_handle_t calculate_distance_handle;
typedef struct{
// se definen las variables
} program_data;
int main() {
program_data pd__attribute__((aligned(16)));
// alineacion para transferencia y
// luego ingresar los datos
// Crear la tarea SPE
speid_t spe_id = spe_create_thread(0,
&calculate_distance_handle, &pd, NULL, -1, 0);
...
spe_wait(spe_id, NULL, 0);
return 0;
}
```

```
#include <spu_mfcio.h>
typedef struct{
// se definen las mismas variables
} program_data;
int main(unsigned long long spe_id, unsigned long
long program_data_ea, unsigned long long env) {
  mfc_get(&pd, program_data_ea, sizeof(pd),
tag\_id, 0, 0);
// Espera para completar
mfc_write_tag_mask(1<<tag_id);
mfc_read_tag_status_any();
// Parte que procesa los datos
...
```

```
mfc_put(&pd, program_data_ea,
sizeof(program_data), tag_id, 0, 0);
mfc_write_tag_mask(1<<tag_id);
mfc_read_tag_status_any();
return 0;
}
```

Estas herramientas desarrolladas para el Cell BE, son actualmente las herramientas estándares para la computación de alto desempeño en diferentes laboratorios del mundo usando los *cell blades* de la IBM. Pero el uso de las PS3 para instalar el linux y las herramientas xlf y xl c/c++ han sido cortadas por la SONY mediante una actualización del firmware de la PS3 que impiden instalar el Linux, lo que ha originado una migración de mucha gente hacia los procesadores gráficos, GPU, usando las herramientas CUDA desarrolladas por la Nvidia.

CUDA

Compute Unified Device Architecture, CUDA, es una arquitectura de cómputo en paralelo desarrollada por la Nvidia para sus procesadores gráficos [5].

Los procesadores gráficos de la Nvidia han sido construídos para realizar cálculos de manera independiente a los procesadores de la(s) placa(s) madre(s). Estos poseen sus propias unidades de memoria de gran porte como los de la placa madre (DDR3-DDR5) entre 1 GB y 6 GB. Así como sus memorias caché L2 de 64kB para cada núcleo. Y aquí viene lo interesante, poseen 48 núcleos o coprocesadores matemáticos para el modelo GeForce GT 520M y 448 núcleos para el modelo Tesla C2075.

El CUDA es un conjunto de herramientas que han permitido crear, semejante a la filosofía del OpenMPI, un conjunto de instrucciones que puedan ser entendidas por el compilador gcc para direccionar las instrucciones entre el procesador de la placa madre y el procesador gráfico y viceversa, para adquirir los datos generados por los núcleos.

La desventaja es que no funciona con el icc de la Intel y tampoco existe una conexión para ser usada con el gfortran.

La técnica de programación es un poco más ardua de realizar porque como se tiene un conjunto grande de núcleos, es necesario agruparlo por categoría o crear un *cloud* con dichos núcleos.

La Nvidia ha creado un conjunto de códigos de muestra que pueden ser usados para crear sus propias herramientas [17]. Aunque el BLAS (*Basic Linear Algebra Subprograms*), FFT (*Fast Fourier Transform*), SPARSE y un generador de números aleatorios han sido incorporados al CUDA a través de librerias ubicadas en el directorio `/usr/lib64`, existen herramientas como el LAPACK (*Linear Algebra Package*) bastante utilizada en la comunidad de física que aún no han sido incorporadas. Y el gran problema es que dichas herramientas están en Fortran. Por ello, hay una gran tarea de establecer dichas herramientas de Fortran para ser utilizadas con el CUDA. Hay una opción comercial desarrollada por la *Portland Group Inc.* [16] para un compilador Fortran que se comunica con el CUDA.

Cuando el CUDA ha sido instalado y el driver de la placa gráfica ha sido activada, el sistema responde de la manera siguiente

```
pablo@blackbird[503]:./C/bin/linux/release/deviceQuery
[deviceQuery] starting...

./C/bin/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "GeForce GT 520M"
  CUDA Driver Version / Runtime Version          4.2 / 4.1
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073414144 bytes)
  ( 1) Multiprocessors x ( 48) CUDA Cores/MP:    48 CUDA Cores
  GPU Clock rate:                                1480 MHz (1.48 GHz)
  Memory Clock rate:                             800 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 65536 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
```

```
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
       with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.2, CUDA Runtime Version = 4.1, NumDevs = 1,
Device = GeForce GT 520M
[deviceQuery] test results...
PASSED

> exiting in 3 seconds: 3...2...1...done!
```

Se observa que la placa gráfica posee 48 núcleos dedicados solamente a realizar cálculos con memoria L2 de 64 kB cada uno. Si uno requiere invertir o diagonalizar matrices, se debe usar algoritmos para reducir a las cuatro operaciones básicas de suma, diferencia, producto y división, porque la cantidad de memoria es muy pequeña para colocar vectores de dimensiones mayores.

Conclusiones

Hemos dado una breve revisión a las posibilidades de realizar programación en paralelo en sistemas de bajo costo. Desde los 2, 4, 6 núcleos que ofrece los procesadores Intel, los 9 núcleos que ofrece los procesadores Cell BE de la Sony-Toshiba-IBM hasta los 48-448 que ofrecen las placas gráficas de la Nvidia se tiene una gran oportunidad de realizar cálculos a muy bajo costo. El gran trabajo que queda por realizar es actualizar las grandes librerías del Fortran al CUDA y al Cell BE. En el caso de los procesadores Intel no es necesario tal tarea titánica porque el *Math Kernel Library* desarrollado por la propia intel basado en las librerías LAPACK y otras del NETLIB [18], y optimizadas para los procesadores Intel en MPI y OpenMP, permite utilizar directamente estas herramientas con los compiladores icc (C/C++) y ifort (Fortran) de la propia Intel.

Existe una gran desventaja en el uso individual de los núcleos, es la poca memoria que disponen los núcleos. Para contrarrestar esta desventaja se usan las funciones `ALLOCATABLE`, que crean espacios de memoria dinámicos antes de un cálculo para un vector o una matriz y después los borra, pero la latencia de los núcleos aumenta disminuyendo la performance total. En este caso se debe establecer un compromiso entre usar las funciones `ALLOCATABLE` para vectores y matrices de modo que el tiempo de todo el proceso sea menor al tiempo de realizar el mismo proceso usando un solo núcleo pero disponiendo de toda la memoria L2. En los procesadores gráficos se hace uso de las funciones `cuda_Malloc` y `cudaMemcpy` para que crear espacios de memoria dinámica y transferir datos en ambas direcciones entre el procesador principal y el procesador gráfico, respectivamente.

Actualmente existen sistemas que combinan procesadores Xeon con procesadores gráficos de la Nvidia [6].

En una computadora personal se puede correr virtualmente dos o más sistemas operativos, todos ellos comunicados virtualmente mediante una red virtual. En un sistema operativo se puede correr el CUDA usando los procesadores gráficos y en otro puede usar las herramientas desarrolladas por la Intel usando los procesadores i7 o Xeon. Aún más se pueden usar varias máquinas con las mismas características con procesadores i7 o Xeon y con dos placas gráficas Nvidia y comunicarlas con el OpenMPI. Esto significa combinar el OpenMP, CUDA y el OpenMPI en un conjunto de máquinas conectadas a través de un ruteador dedicado, un *cluster*.

Agradecimientos

Agradecemos el soporte financiero del Consejo Superior de Investigaciones perteneciente al Vicerrectorado de Investigación de la Universidad Nacional Mayor de San Marcos mediante los contratos N.°s 101301021 y 111301031 que nos ha permitido contar con los procesadores i7-980X y GeForce GT 520M, y a Nazgul Tec por permitirnos el uso del procesador Cell BE.

Referencias

[1] http://en.wikipedia.org/wiki/Unix

[2] http://www.openmpi.org,
http://en.wikipedia.org/wiki/Open_MPI

[3] http://www.openmp.org,
http://en.wikipedia.org/wiki/OpenMP

[4] http://en.wikimedia.org/wiki/wiki/Cell_(microprocessor),
http://www.ibm.com/developer/power/cell

[5] http://en.wikipedia.org/wiki/CUDA,
http://developer.nvidia.com/nvidia-gpu-computing-documentation

[6] http://www.top500.org,
http://en.wikipedia.org/wiki/TOP500

[7] http://en.wikipedia.org/wiki/Network_File_System

[8] http://en.wikipedia.org/wiki/Network_Information_Service

[9] http://www.pnfs.com

[10] http://en.wikipedia.org/wiki/Thinking_Machines_Corporation

[11] http://en.wikipedia.org/wiki/Cray

[12] http://en.wikipedia.org/wiki/NCUBE

[13] http://en.wikipedia.org/wiki/Hyper-threading

[14] http://en.wikipedia.org/wiki/Reduced_instruction_set_computing

[15] http://www-01.ibm.com/software/awdtools/fortran/

[16] http://www.pgroup.com

[17] http://www.nvidia.com/object/cuda_home_new.html

[18] http://www.netlib.org