Programación de Alto Desempeño

Rodríguez-León,Abelardo[1]

[1] Instituto Tecnológico de Veracruz. M.A. De Quevedo 2779, CP91860, Veracruz, Ver. Mexico.
arleon@itver.edu.mx

Resumen. El aprovechamiento del poder de computo del hardware moderno ha cambiado. Ahora es mas común encontrar cluster de computadoras donde cada una de ellas puede ser multiprocesador, los procesadores pueden ser multinucleos y cada núcleos soporta paralelismo por extensiones multimedia. Lo anterior da por resultado muchos niveles en los que el programador puede aplicar diversas técnicas al momento de realizar programas. En este articulo se hace una revisión de las principales tendencias en el desarrollo de programas con altos requerimientos de computo. Se describe el estándar usado en cada nivel y se da al final una prospectiva de como serán las cosas en el futuro.

Palabras clave: Programación paralela, MPI, OpenMP, extensiones multimedia

1 Introducción

La limitación física en el aislante que separa los transistores en los actuales microprocesadores ha limitado el incremento de la velocidad de los mismo, como se venia dando hace algunos años. Por otra parte el avance en la miniaturización de componente electrónicos no se ha detenido, dando por resultados microprocesadores mas pequeños con menor consumo de energía.

Estas dos característica dieron por resultado un cambio en la forma que los fabricantes de microprocesadores para el usuario común, han aumentado la capacidad de computo. El resultado fue la posibilidad de poner mas de un núcleo de procesamiento dentro de la misma pastilla del procesador, dando lugar a la tecnología llamada multinucleo. El lanzamiento se dio en el 2006 cuando Intel y AMD sacaron al mercado procesadores multinucleo. Anteriormente compañías como IBM y SUN había sacado ya maquinas multiprocesadores pero dirigidas a un mercado mas especializado por el alto precio de las mismas.

Por otro lado el abaratamiento del hardware en general y el desarrollo de software libre ha permitido la creación de clusters de bajo costo comúnmente conocidos como Beowulf [1]. En el proyecto Beowulf se desarrollaron metodologías para utilizar configuraciones de hardware-software de bajo costo en problemas de alta exigencia

M.A Cruz-Chávez, J.C. Zavala-Díaz (Eds): CICos 2008, ISBN:978-607-00-0165-9. pp. 63-76, 2008

de cómputo. El sistema operativo utilizado para este proyecto es Linux. La figura 1 muestra un esquema de conectividad de un clúster de tipo Beowulf.

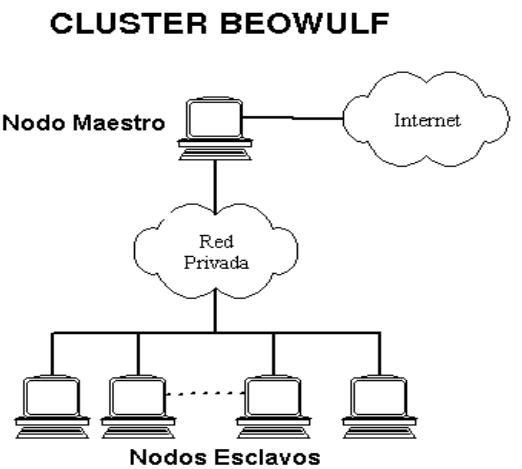

Figura 1. Estructura general de un cluster de computadoras de tipo Beowulf.

En México existen varias implementaciones de cluster de computadoras tipo Beowulf, siendo las más importantes Kan Balam en DGSCA-UNAM con 1368 procesadores y la Cray XD1 del CNS con 216 procesadores. En el Instituto Tecnológico de Veracruz se han desarrollado varios prototipos de cluster tipo Beowulf, el primero de ellos llamado NOPAL se hizo en el 2003 con 8 computadoras Pentium I de 133 Mhz y se actualizado en el 2007 con 4 computadoras Pentium III de 600 Mhz.

Las maquinas que forman un cluster deben funcionar como una sola computadora, si se preparan los programas adecuadamente, lo cual no es una tarea trivial. Es de esperar que en un futuro esta situación cambie, para lo cual se requieren importantes contribuciones en investigación y desarrollo, análisis y diseño de algoritmos paralelos, diseño de compiladores, modelos de programación, lenguajes paralelos, optimización de código y paralelización automática.

El objetivo de este articulo es dar una revisión de las principales tecnología de software que permitan aprovechar los diversos niveles de computo que se tienen en un cluster de estaciones trabajo.

2 Métodos de paralelización

Las áreas científicas y tecnológicas actualmente tratan de resolver problemas que demandan, en muchos casos, una gran capacidad de cómputo. Algunos ejemplos son:

Procesamiento de imágenes para navegación automática, Diseño asistido por computadora, Diagnóstico médico, Inteligencia artificial, Modelos de simulación, Ingeniería genética, etc.

La alternativa más interesante hoy en día para satisfacer los requerimientos de potencia de cálculo de estas aplicaciones es la utilización simultánea de múltiples elementos de proceso. Hay un amplio espectro de posibilidades en función de la granularidad, de los elementos que componen el problema y de la forma como interactúan.

En un extremo están las arquitecturas con elementos de proceso simples que operan bajo un control centralizado, generalmente actúan como coprocesadores gobernados por la CPU central (p.ej. extensiones multimedia). Una opción intermedia es la utilización de unidades de proceso más autónomas en la que se ejecutan simultáneamente subprocesos pertenecientes a un proceso que ejecuta la CPU central (p.ej. Hyperthreading). Finalmente, cada elemento de proceso puede ser totalmente autónomo con lo que puede ejecutar procesos totalmente independientes, es el caso de los multiprocesadores (p.ej. procesadores multinucleo) y los multicomputadores (p.ej. redes de PCs).

Los modos de interacción entre los elementos de proceso van desde la utilización de un control centralizado (arquitecturas SIMD), pasando por la utilización de memoria compartida (multiprocesadores), hasta la utilización de redes de interconexión (multicomputadores).

Las múltiples alternativas mencionadas no son disjuntas sino que se complementan. Es posible utilizar clusters de PCs en los que cada nodo es un multiprocesador y, a su vez, cada procesador de un nodo dispone de extensiones multimedia. El hecho de que en la actualidad pueda disponerse de este tipo de entorno a bajo coste hace que sea la opción más utilizada en aplicaciones prácticas.

A continuación se describiran brevemente las técnicas de programación utilizadas en clusters en todos sus niveles. Dichas técnicas comprenden: paralelismo con paso de mensajes, paralelismo multihilo y paralelismo SIMD.

3. Paralelismo con paso de mensajes: MPI

La descripción de más alto nivel de un cluster se corresponde con computadoras autónomos conectados mediante una red. La forma natural de programación a este nivel consiste en procesos independientes que colaboran intercambiando mensajes.

Desde la aparición de los primeros multicomputadores se han desarrollado múltiples métodos de programación con pasos de mensajes como Occam[2], PVM[3] y MPI[4]. Entre todos ellos MPI se ha convertido en un estándar ampliamente utilizado y soportado en prácticamente todas las plataformas disponibles.

MPI (Message Passing Interface) consiste en un entorno de ejecución y una librería de programación. El entorno de ejecución ofrece los servicios requeridos para acceder al soporte hardware de comunicación y la librería proporciona la interfaz de programación para iniciar los procesos y realizar las comunicaciones. El entorno de

ejecución y su interfaz puede diferir ligeramente entre las diferentes implementaciones, sin embargo la interfaz de programación está totalmente estandarizada de forma que los programas basados en MPI son totalmente portables.

Un aspecto que puede diferir bastante de una implementación a otra es el nivel de prestaciones obtenido. Este es un aspecto a estudiar de forma detenida para seleccionar la implementación de MPI más conveniente sobre una plataforma determinada.

Las características principales de la interfaz de programación que proporciona MPI son comunicación punto a punto síncrona y asíncrona, comunicaciones colectivas y topologías virtuales. La versión 2 del estándar añade capacidad multihilo, creación dinámica de procesos, entrada-salida paralela y comunicación unidireccional. Este último aspecto es particularmente novedoso ya que aproxima MPI al modelo de memoria compartida. Se trata de que un proceso pueda acceder a variables ubicadas en otro proceso sin que este último intervenga.

3.1. Características generales.

Las implementaciones de MPI están disponibles principalmente para el lenguaje C. Las extensiones para C++ y las implementaciones para Fortran son bastante comunes. En la actualidad también se pueden encontrar implementaciones para Java.

En el caso del lenguaje C, el formato de las funciones MPI es el siguiente:

rc = MPI_Xxxxx(parámetros);

Donde rc es una variable tipo entero que retorna el valor de cero o uno según el éxito/fracaso de la función a su término. Se tiene una llamada a alguna rutina que debe comenzar con la palabra MPI en mayúsculas, seguida de un carácter de subrayado. Seguidamente aparece el identificador de la función que comienza con mayúscula y continúa con letras minúsculas o guiones subrayados.

Estructura básica de un programa paralelo.

Un programa paralelo que utilice MPI tendrá que realizar las siguientes operaciones:

1. Inicializar el entorno de MPI.
2. Comunicación entre procesos.
3. Finalizar el entorno de MPI.

La versión 1.0 del estándar MPI define 125 funciones. Sin embargo, las funciones básicas, para realizar las operaciones anteriores, son las seis que se pueden observar en la tabla 1.

Tabla 1. Funciones básica de MPI.

Tipo de función	Nombre	Acción
Inicialización del entorno:	MPI_Init () MPI_Comm_size ()	Inicializa el entorno de MPI. Retorna el número de procesos.

	MPI_Comm_rank ()	Retorna el identificador de cada proceso.
Comunicación	MPI_Send () MPI_Recv ()	Envía un mensaje Recibe un mensaje
Finalización del entorno	MPI_Finalize()	Termina el entorno de ejecución paralela.

Comunicadores y topologías

El envío de mensajes entre procesos MPI se realiza por medio de comunicadores. Un comunicador es una colección de procesos que comparte el envío-recepción de mensajes. Por defecto MPI crea un comunicador general que incluye todos los procesos disponibles en el entorno paralelo, llamado MPI_COMM_WORLD.

MPI proporciona funciones que permiten al usuario (a partir de este comunicador general MPI_COMM_WORLD) crear comunicadores con distintos procesos. Los comunicadores se usan para restringir los mensajes a los procesos dentro del comunicador. Los mensajes enviados entre procesos de un comunicador pueden ser punto a punto o colectivos.

Cada proceso tiene asociado un número único (rank) en el contexto del comunicador. Por ejemplo, en el contexto del comunicador global (MPI_COMM_WORLD) se numeran desde cero hasta n-1. En el contexto de un comunicador creado por el usuario, cada proceso del grupo recibe un nuevo rank, relativo al número de procesos pertenecientes al comunicador.

Una forma de organizar el acceso a los procesos que forman parte de un comunicador, es por medio de las topologías. Una topología es una estructura virtual, basada en el comunicador, que permite acceder organizadamente a los procesos dentro del comunicador.

MPI define dos tipos de topologías: cartesiana y grafo. La cartesiana consiste en una malla que identifica los procesos mediante vectores de índices correspondientes a un arreglo cartesiano multidimensional de procesos. La topología grafo es una organización más general, donde los procesos son identificados y accedidos mediante un grafo previamente definido.

MPI cuenta con un amplio conjunto de funciones para manejar tanto comunicadores como topologías basadas en comunicadores.

Comunicaciones colectivas

La comunicación colectiva involucra a todos los procesos que pertenecen al mismo comunicador [5]. Algunas de las características de la comunicación colectiva son [4]:

- Una comunicación colectiva actúa como barrera de sincronización para los procesos involucrados.
- Un mensaje es un arreglo de un tipo específico de dato.
- Las funciones usadas para una comunicación colectiva, utilizan 2 parámetros para declarar el tipo de dato que se distribuirá en los procesos.

Diferencias con las comunicaciones punto a punto:

- No existe el concepto de etiqueta.
- El envío del mensaje debe coincidir con las especificaciones del buffer de recepción.

Las funciones que MPI provee para comunicaciones colectivas son:

Función	Acción
MPI_Bcast()	Un solo proceso (raíz) envía el mismo dato a todos los procesos del comunicador actual.
MPI_Scatter()	Un solo proceso (raíz) envía un arreglo de datos a los procesos del comunicador actual. Cada elemento del arreglo es enviado a un proceso diferente. El tamaño del arreglo debe ser igual al número de procesos en el comunicador.
MPI_Gather()	Todos los procesos envían datos al proceso inicial. Es la rutina contraria a Scatter.
MPI_Allgather()	Similar a Gather, pero ahora los datos son recibidos en todos los procesos.
MPI_Alltoall()	Cada proceso distribuye un arreglo de datos (de igual número que procesos) a todos los demás procesos del comunicador.
MPI_Reduce()	Realiza en el nodo raíz alguna de las operaciones de reducción validas, con los datos recibidos de todos los demás procesos, del comunicador.

MPI_Barrier provee un mecanismo para sincronizar todos los procesos de un comunicador sin envió de datos. Cada proceso hace una pausa mientras todos los procesos del comunicador son llamados por MPI_Barrier.

4. Paralelismo multihilo: OpenMP

En la actualidad los nodos que constituyen un cluster son comúnmente multiprocesadores y cada procesador generalmente es multinucleo. Con esta arquitectura también se puede utilizar MPI, pero el modelo más natural de programación paralela es la programación multihilo. Hay básicamente dos aproximaciones: Soporte multihilo en el lenguaje o el compilador y Soporte mediante librería de programación.

En el caso del lenguaje C se dispone de OpenMP para el soporte multihilo en el compilador y del estándar POSIX para librerías multihilo. OpenMP es un API estándar basado principalmente en el uso de directivas (pragmas), que definieron los principales fabricantes de hardware y software, para permitir el desarrollo de aplicaciones paralelas con memoria compartida.

OpenMP incluye directivas y funciones que permiten definir regiones paralelas, trabajo compartido, sincronización y manejo de datos comunes. Tuvo su origen en el estándar ANSI X3H5, que fue el primer intento de estandarizar el manejo de paralelismo por medio de hilos [6].

OpenMP desplaza al estándar POSIX [7] en la preferencia de los programadores, debido a que cuenta con una serie de esquemas predefinidos de paralelización y además facilita la definición, el control y sincronización de los hilos. En particular, la paralelización más común, que es la paralelización de bucles, resulta muy sencilla.

A pesar de proporcionar esquemas predefinidos que facilitan la paralelización, esto no garantiza que se haga de forma efectiva. Para ello habrá que realizar una exhaustiva identificación de cuellos de botella y una sintonización de la implementación (número de hilos, tipo de planificación, etc).

OpenMP está disponible para Fortran y para C/C++. La ultima especificación del estándar es la 3.0 de mayo del 2008 [8], aunque los compiladores disponibles (como el de Intel C++ 10) soportan generalmente la versión 2.5 de Mayo del 2005 [9].

Modelo de Programación

OpenMP establece 3 tipos de regiones paralelas, las cuales se pueden ver en la figura 2.

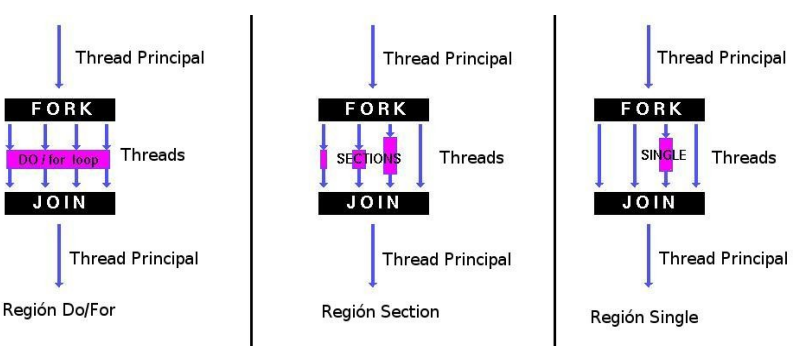

Figura 2. Regiones paralelas en OpenMP

- **Do/For.** Es una región que se ejecuta en un bucle, repartiendo las iteraciones totales entre los hilos disponibles. Esto hace que cada hilo ejecute una porción del ciclo. Este tipo de región está pensada para el caso en el que las operaciones dentro de un ciclo tienen cierto nivel de independencia entre ellas
- **Sections**. Es una región donde se define un trabajo específico para cada hilo, haciéndose una sincronización al final, cuando todos han terminado para continuar el trabajo. Este tipo de región está pensado cuando se requieren hacer varias tareas diferentes a la vez.

- **Single**. Es una región en la cual solo se ejecutará el trabajo indicado por un hilo. Esta directiva está pensada para ser usada dentro de alguna de las otras dos para hacer alguna tarea que tenga una dependencia como una E/S.

Las regiones arriba citadas pueden combinarse entre sí, aunque es necesario tener cuidado para evitar crear una sobrecarga excesiva. La granularidad de las tareas ha de ser lo suficiente grande para amortizar la sobrecarga de creación, finalización y sincronización de los hilos. Por otra parte, no tendrá sentido definir hilos cuando la sincronización entre ellos provoque su serialización.

Esquemas de sincronización

La colaboración de múltiples hilos dentro de un proceso requiere disponer de mecanismos de sincronización. Existen seis esquemas de sincronización en OpenMP [6]:

- **Master**. Establece que solo el hilo principal es el que ejecuta la región que se indica en esta directiva, los demás hilos no lo hacen.
- **Critical.** Establece que la región indicada en esta directiva no puede ser accedida por más de un hilo al mismo tiempo.
- **Barrier.** Establece un punto de espera para sincronizar todos los hilos. Una vez que todos han llegado a la barrera entonces continúan con la ejecución.
- **Atomic.** Establece una mini sección de tipo Critical en la que se determina que una región de memoria debe ser accedida por un hilo a la vez.
- **Flush.** Establece un punto en el cual todos los hilos que usen ciertas variables de memoria, actualizan el contenido de las mismas.
- **Ordered.** Establece en una región Do/For ciertas instrucciones que se ejecutan como si fueran secuenciales.

5. Modelos Híbridos: MPI-OpenMP

Como ya se comento antes, las plataformas actuales combina la arquitectura de memoria distribuida (a nivel global) con la de memoria compartida (a nivel de cada nodo). Surge pues de forma natural la idea de utilizar de forma combinada los dos modelos de programación mencionados anteriormente: paso de mensajes con MPI y paralelismo multihilo con OpenMP. De hecho en los últimos años se han desarrollado modelos híbridos de programación paralela [10] [11] que permiten aprovechar -en un mismo problema- lo mejor de ambos esquemas:

Hibrid-PC. Es un modelo que permite la comunicación directa de proceso a proceso. En este modelo, OpenMP se usa dentro de cada nodo SMP y las comunicaciones MPI se realizan fuera de la región paralela.

Hibrid-TC. Es un modelo similar al anterior pero aquí se designa un hilo que se encarga de las comunicaciones MPI, los demás hilos se dedican al cálculo.

Tareas Orientado al Solapamiento. Este es un modelo que permite hacer que los hilos de ejecución sean independientes del hilo que llevará las comunicaciones.

Planificador Hiperplano. Este modelo considera dar tratamiento diferente a los hilos que pueden ejecutarse concurrentemente sin dependencia de datos, de los que tienen alguna dependencia o comunicación MPI.

Estos modelos han demostrado [10],[11],[12] dar buenos resultados en determinados tipos de problemas. Sin embargo, estas soluciones no son ni triviales ni universales, esto quiere decir que no son sencillas de implementar, debido a la cantidad de combinaciones de dependencias de cálculo y de comunicaciones que se pueden presentar.

6. Paralelismo SIMD: Librería Intrinsics

Algo poco aprovechado en los microprocesadores actuales es el soporte hardware y de instrucciones en el desarrollo de aplicaciones que usen programación SIMD (Single Instruction, Multiple Data). Las programación SIMD dentro del procesador se basa en el uso de los registros adicionales que tiene internamente el procesador para colocar en ellos datos que después serán procesados en un sola instrucción en un ciclo maquina, como se puede observar en la figura 3. Esto es un tipo de paralelismo de grano muy fino ya que funciona prácticamente a nivel de instrucción, realizando en una misma instrucción, operaciones sobre diferentes datos.

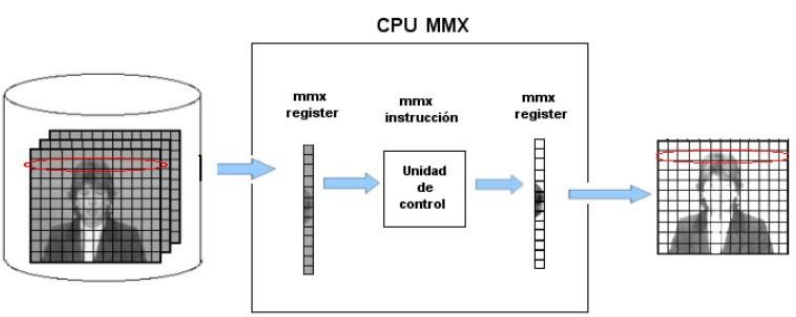

Figura 3. Ejemplo de funcionamiento de extensiones multimedia.

La mejor forma de usar esta capacidad es por medio de código directo en ensamblador dentro del código fuente de C o C++. Sin embargo, hacer esto puede llevar demasiado tiempo, resultar tedioso y es muy propenso a errores. Además, la inserción de código ensamblador no está soportada por todos los compiladores.

Como alternativa, hay dispone de librerías de funciones en C que ocultan la implementación en ensamblador. Una de ellas es la librería Intrinsics [13],[14] junto con el compilador de Intel.

Intrinsics está constituida por funciones que generan código ensamblador de acceso a las extensiones SIMD (instrucciones MMX, SSE y SSE2) y al mismo tiempo

permiten al programador usar variables y llamadas a funciones según la sintaxis de C/C++. Al invocar una función Intrinsic, esta es expandida en línea (inline), esto elimina la sobrecarga por llamada a función. El resultado es muy cercano a poner llamadas a ensamblador en línea que no interfiere con las optimizaciones del compilador. Mientras que muchas de las funciones Intrinsics corresponden directamente a instrucciones en ensamblador, otras son compuestas, formadas por llamadas a varias instrucciones ensamblador.

Así pues, mediante Intrinsics se accede de forma cómoda a la implementación de computación paralela que no puede ser realizada usando solo C/C++ estándar. Paralelizar código puede reducir significativamente el tiempo de ejecución de los métodos que requieren cómputo intensivo. Más aún, usando Intrinsics (en lugar de código ensamblador) se habilita al compilador para realizar una mejor optimización de la planificación de las instrucciones. Si no se codifica completamente una rutina en ensamblador, es difícil alcanzar un nivel de eficiencia igual al que el compilador (en conjunto con las rutinas de Intrinsics) puede lograr.

Extensiones multimedia

La arquitectura SIMD para computación paralela ha sido incluida en los últimos años en los microprocesadores para computadoras personales [15],[16],[17]. La carrera comenzó en 1997 cuando Intel presentó su procesador con soporte multimedia llamado Pentium MMX (MultiMedia eXtension).

MMX es un conjunto instrucciones que agrega 8 nuevos registros en la arquitectura interna del procesador (MM0 a MM7). En realidad estos registros solo renombran registros ya existentes en la unidad de punto flotante (FPU, Float Point Unit) X87, con lo que las aplicaciones que hagan uso de ambos tipos de registros deberán indicar explícitamente que usarán los registros para punto flotante o MMX. Esto complica la programación y puede ralentizar la ejecución -en un programa que combine ambos esquemas- por cambios de contexto.

Cada registro de MMX es de 64 bits y puede contener datos de 64, 32, 16 y 8 bits, empaquetados en dicho registro. Los registros de FPU son de 80 bits con lo que los registros MMX no usan los 16 bits más altos de los registros FPU.

Las 57 instrucciones de bajo nivel que soporta MMX manejan operaciones de tipo entero en operandos de 64 bits definidos sobre los registros FPU.

En 1999, Intel presentó SSE (Streaming SIMD Extensions) en la serie de procesadores Pentium III, como una respuesta a la introducción por parte de AMD de la tecnología 3DNow. Posteriormente AMD agregó soporte para instrucciones SSE a partir del procesador Athlon XP.

SSE contiene 70 nuevas instrucciones y resuelve los principales inconvenientes que tiene la tecnología MMX: Compartir registros MMX con la FPU y manejar instrucciones enteras.

SSE agrega 8 registros nuevos de 128 bits, conocidos como XMM0 a XMM7. Cada registro empaqueta juntos 4 flotantes simples de 32 bits.

Debido a que estos 8 registros de 128 bits son adicionales a los registros existentes, el sistema operativo debe preservar la integridad de las tareas, habilitando explícitamente el uso de estos registro.

La característica de que SSE soporte operaciones con flotantes, hace que sea más usado en la actualidad que MMX, sobre todo ahora que las tarjetas gráficas soportan internamente operaciones enteras. Las operaciones SIMD enteras pueden ser realizadas con los registros MMX de 64 bits.

El primer procesador con soporte SSE, el Pentium III comparte los recursos de ejecución entre SSE y FPU. Esto quiere decir que las instrucciones FPU y SSE no se pueden llevar a cabo en el mismo ciclo de procesador, a pesar de usar diferentes registros. Por otro lado, el manejar registros separados tiene la ventaja de que no requiere intercambios de datos en los registros para preservar el estado del sistema, lo cual mejora el rendimiento.

Intel introdujo SSE2 en el 2001 con el procesador Pentium 4. SSE2 [18] es un conjunto de instrucciones que extiende las características de SSE y que pretende sustituir por completo a MMX.

SSE2 define también 8 registros de 128 bits y además agrega 144 nuevas instrucciones a las 70 que proporcionaba SSE. Dentro de las instrucciones agregadas están las que permiten manejar operaciones SIMD enteras en 128 bits. Con esta característica, resulta ya innecesario manejar las instrucciones MMX con sus 8 registros de 64 bits. SSE2 permite hacer operaciones entre enteros y flotantes sin el cambio de contexto necesario entre los registros MMX y los FPU.

Otras instrucciones de SSE2 permiten controlar la caché previniendo la contaminación de la misma, -cuando se procesan flujos indefinidos de información- y un sofisticado conjunto de instrucciones de conversión de formatos numéricos.

AMD implementó SSE2 en plataformas AMD64 incluyendo 8 registros más para un total de 16 (XMM0 a XMM15). Los registros adicionales son visibles solo cuando se está trabajando en modo 64 bits. Intel adoptó en el 2004 estos 8 registros adicionales como parte de su soporte a la arquitectura AMD64 (renombrada por Intel como EM64T).

Diferencia entre x87 FPU y SSE2 para manejo de flotantes.

La FPU almacena los resultados en sus registros con 80 bits de precisión. Cuando se pasa el software a SSE2, la combinación de operaciones matemáticas o tipos de datos pueden derivar en desviaciones numéricas.

El problema ocurre debido a que el compilador traduce algunas expresiones matemáticas en una secuencia de operaciones básicas. Dependiendo del compilador y las optimizaciones usadas, una expresión matemática puede necesitar diferentes resultados intermedios almacenados temporalmente y reutilizados después. Estos resultados pueden variar por truncamiento de la FPU al pasar de 80 a 64 bits.

Diferencias entre MMX y SSE2.

SSE2 soporta todas las operaciones enteras de MMX, por ello es posible convertir cualquier código MMX a su equivalente en SSE2. Como un registro XMM es dos veces más grande que un registro MMX, las variables de control de ciclos y los accesos a memoria pueden requerir algunos cambios.

Aunque una instrucción SSE2 puede operar con varios datos, en forma similar a como ocurre con una instrucción MMX, el rendimiento puede no incrementarse significativamente. Hay dos razones para ello: el acceso a datos en memoria no alineados a 16 bits trae consigo una penalización y la productividad de instrucciones SSE2 en muchas implementaciones X86 es usualmente más pequeña que las instrucciones MMX.

Modos de Intrinsics

Muchas de las instrucciones MMX, SSE y SSE2 tienen su correspondiente función implementada directamente en Intrinsics. Como la arquitectura Itanium no soporta paralelismo ni doble precisión, SSE2 no es implementado en estos sistemas. Aunque muchas funciones de Intrinsics son provistas para incrementar el rendimiento, algunas son implementadas solo por portabilidad y pueden degradar el rendimiento. La principal razón para ésta degradación es que algunas instrucciones no son soportadas directamente en algunas plataformas. Así, podemos distinguir entre funciones Intrinsic que mejoran el rendimiento y aquellas que solo son para compatibilidad. Algunas funciones Intrinsics pueden generar código incompatible con algunos procesadores. Por ello es responsabilidad del programador generar el código apropiado para cada procesador.

Las funciones Intrinsics tienen que definir dentro de ellas compatibilidad entre los tipos de datos que se manejan en las diferentes extensiones multimedia que define Intel. Por ejemplo, en MMX los registros son de tipo __m64. Si usamos una función que utiliza instrucciones MMX, y posteriormente usamos otra función que utiliza funciones SSE; entonces necesitamos “vaciar el estado de multimedia” con la función _mm_empty previamente. Los prototipos de las funciones Intrinsics están en el archivo de cabeceras mmintrin.h .

El tipo de datos para SSE es _m128 y puede operar con 4 valores flotantes de simple precisión. Cada valor de _m128 es almacenado en la porción más significativa de cada par de registros flotantes. Esto es, en una instrucción Intrinsic los operandos están compuestos y se corresponden a un par de registros de punto flotante. En la mayoría de las arquitecturas de Intel, las operaciones como sumas y comparaciones se hacen con dos instrucciones SIMD. Sin embargo ambas pueden ser realizadas en el mismo ciclo de reloj, a través de una operación básica SSE, o cuatro operaciones de punto flotante de simple precisión por ciclo.

Cuando se utilizan instrucciones SSE Intrinsics de punto flotante, el programador ha de estar familiarizado con lo que el hardware le permite hacer a SSE. Hay dos consideraciones importantes a tener en cuenta:

- Las instrucciones compuestas pueden degradar el rendimiento.
- Al cargar o almacenar datos de punto flotante como objetos _m128, los datos han de ser alineados a 16 bytes. En general, una referencia a n bytes de datos puede ser alineada en torno a n bytes, para evitar penalización en el rendimiento.

SSE2 Intrinsics funciona con valores que van desde un valor en punto flotante de doble precisión hasta 16 enteros de 8 bits.

7. Conclusiones.

Basado en el recorrido tecnológico que se ha dado en los puntos anteriores, podemos darnos cuenta que actualmente la tarea de crear aplicaciones paralelas eficientes no es sencilla, debido los niveles y a la discrepancia que hay en algunos de ellos debido a que los fabricante no han terminado de ponerse de acuerdo.

En los niveles de paralelización de alto nivel (de grano grueso y medio) el panorama es relativamente claro. MPI se consolidara como estandar en paralelización de grano grueso y OpenMP como el estandar en paralelismo de grano medio. En estos dos niveles veremos cual de los dos estandar prevalece ya que tanto MPI como OpenMP en un futuro cercano empezarán a agregar características que permitan realizar paralelismo en el otro nivel . MPI esta ya proporcionando manejo de threads y OpenMP eventualmente incluirá rutinas que permitan acceder a datos en programas localizados en otros nodos. Existen entonces dos posibilidades, una, que alguno de los dos estándares cubra lo que el otro proporciona, lo cual dará por resultados el desplazamiento del menos desarrollado. La otra posibilidad es que ambos se unan y creen un solido estándar que permita realizar programación paralela de grano grueso y medio a la vez.

En el nivel de grano fino (extensiones multimedia), la cuestión es diferente ya que ahí se depende mucho del procesador, y por tanto del soporte que den los fabricantes al programador para usar el paralelismo de extensiones multimedia. Tendrán que ponerse de acuerdo ambos para definir una sola interfaz que permita aprovechar el paralelismo de instrucciones sin necesidad de reescribir código especifico a una arquitectura, cosa que se ve poco probable.

La otra alternativa es que se cree un estándar independiente del procesador que defina (como lo hizo OpenMP a nivel de threads) modelos de paralelismo de grano fino independiente del fabricante del procesador.

Independiente de como evolucionen las dos situaciones descritas anteriormente veremos un incremento en el numero de aplicaciones que aprovechen el poder de computo adicional que nos proporciona el hardware actual, por lo que los actuales programadores deberán prepararse para que en el futuro cercano puedan crear este tipo de aplicaciones, usando los estándares ya disponibles.

References

1 Sterling, T., Becker, D. and Saverese, D. Beowulf: A Parallel Workstation for Scientific Computation. Proceedings of International Conference on Parallel Processing, 1995.

2 Uwe Kastens, Friedhelm Meyer, Alf Wachsmann, and Friedrich Wichmann: “Occam-light: A Language Combining Shared Memory and Message Passing”. Proceedings 3. (PASA) Workshop Parallele Systeme und Algorithmen. Gesellschaft fur Informatik, 1993.

3 G. A. Geist, J. A. Kohl, P. M. Papadopoulos: “PVM and MPI: a Comparison of Features”. Calculateurs Paralleles Vol. 8 No. 2 (1996).

4 Pacheco, P.S.: Parallel Programming with MPI, Morgan Kaufman Publishers, Inc, 1996

5 Snir, M. Otto, S. et. al. "MPI: The complete reference". The MIT Press. Cambridge, Mássachusetts,1996.

6 Lawrence Livermore National Lanboratory. “EC3507: Tutorial OpenMP”. http://www.llnl.gov/computing/tutorials/openMP/ , Last Modified: 04/14/2008.

7 Bil Lewis, Daniel J. Berg: “PThreads Primer. A Guide to Multithreaded Programming”. Ed. SunSoft Press, A Prentice Hall Title. 1996.

8 OpenMP Architecture Review Board. “OpenMP Application Program Interface ”. Version 3.0, May2008.

9 OpenMP Architecture Review Board. “OpenMP Application Program Interface ”. Version 2.5, May 2005.

10 Drosinos, N. , Koziris N. : Performace Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Cluster. IPDPS 2004.

11 Quoc T.V., Yoshinaga T. Abderazek B.A., Sowa M.: Construction of Hybrid MPI-OpenMP Solutions for SMP Clusters. IPSJ Transactions on Advanced Computing Systems,46, No.SIG 3(ACS8) 2005.

12 Quoc T. V., Yoshinaga T.: Optimization for Hybrid MPI-OpenMP Programs on a Cluster of SMPs. Symposium on Advanced Computing Systems and Infrastructures, 2004.

13 Intel Co. “Intel C++ Compiler for Linux Systems, User's Guide”. Cap. Intel C++ Intrinsics Reference. 2003.

14 Joseph D. Wieber, Jr and Gary M. Zoppetti: “How to use Intrinsics”. Ed. Intel Corporation. http://softwarecommunity.intel.com/articles/eng/3370.htm. Last modified: 05/16/2008.

15 Intel Co. “IA-32 Intel Architecture Software Developer’s Manual”, Volume 1: Basic Architecture, Cap. 9. Programming with MMX. 2006

16 Intel Co. “IA-32 Intel Architecture Software Developer’s Manual”, Volume 1: Basic Architecture, Cap. 10. Programing with SSE. 2006

17 Bipin Patwardhan: “Introduction to the Streaming SIMD Extensions in the Pentium III: Part I, II and III”. http://www.x86.org/articles/. 2006

18 Intel Co. “IA-32 Intel Architecture Software Developer’s Manual”, Volume 1: Basic Architecture, Cap. 11. Programming with SSE2. 2006.