Computación de Altas Prestaciones

Optimización de código

Optimización de código: Objetivos

- Reducir tiempo de CPU
- Reducir tiempo de E/S
- Reducir tamaño del programa

- Reducir tiempo total de ejecución

Optimización de código: Técnicas a recordar

- Opciones de compilación adecuadas
- Selección de algoritmos (librerías)
- Optimización manual

- Sobre todo, “No hacerlo mal”.

Optimización de código:Cuestiones Generales.

- La optimización debe hacerse “cuando tiene sentido”, valorando el tiempo del programador
- También, al optimizar el programa puede volverse menos “comprensible”, y, por lo tanto, más difícil de depurar.
- Un programa optimizado para ejecución “secuencial” puede ser menos “paralelizable” que el no optimizado
- La optimización entra en conflicto con otras características deseables de un programa

Optimización de código:

- Se deben usar las opciones de optimización automática del compilador:

-O1, -O2, -O3, -O4.

- Todos los compiladores disponen de muchas opciones, que se pueden usar:

-Para tipo concreto de CPU

Optimización de código: Detección de puntos lentos

- Si con las opciones automáticas no es suficiente:

-Detección de los “puntos lentos” tomando tiempos, o, preferiblemente, mediante herramientas de “Profiling”:

-Intel Vtune

-GNU: añadir flag –p, utilizar **gprof**

Optimización de código: "Consumidores de tiempo"

1)Carga de programa (librerías): Uso intensivo de DLLs.

- Alternativa, librerías estáticas (tienen ventajas e inconvenientes)

-En general, siempre será más rápido usar pocos archivos en el mismo directorio que el .exe, que tener muchas dependencias en diferentes sitios del disco duro.

Optimización de código: "Consumidores de tiempo"

2) Acceso a Archivos

- Acceso secuencial es mejor que acceso aleatorio.
- Leer o Escribir grandes bloques, mucho mejor que elemento a elemento.
- Acceder a un archivo recientemente accedido es más rápido (cache de disco).
- Mejor binario que ASCII.
- Si hay que acceder a varias estructuras de datos grandes, mejor acceder una a una que no hacer accesos mezclados.
- Hacer "mirrors" de datos en RAM.
- Si es posible, puede ser útil hacer un "Thread" para I/O de archivos.

Optimización de código: "Consumidores de tiempo"

3) Acceso a otros recursos

- Pantalla, Impresora
- Registro (en Windows)
- Acceso a red

4) Cambios de contexto, Acceso a RAM+caches, cadenas de dependencia, etc.

Optimización de código: Selección de algoritmos

- Librerías disponibles: STL, Intel MKL, LAPACK, BLAS, ... Incluso para tareas aparentemente triviales:

Mal:

```
int y[N], x[N]
...
for(i=0;i<N;i++)
    Y(i)=X(i);
```

Bien:

```
int y[N], x[N]
...

memcpy(Y,X,N*sizeof(int))
```

OPTIMIZACIÓN MANUAL

Optimización de código (MANUAL)

- Es la última opción
- Debe conocerse, aunque solo sea para no escribir programas “ineficientes”
- Algunas de las técnicas las aplican directamente los compiladores.
- Conviene “no excederse”, para que el programa sea legible.

Optimización de código (MANUAL)

- Optimización de bucles
- Evaluación de expresiones
- Llamadas a subprogramas

Optimización de código (MANUAL)

- Evaluación de expresiones

  Escribir la expresión en la mejor forma, no todas son iguales.

Mal:

```
Y=C[1]+C[2]*X+C[3]*pow(X,2)
+C[4]*pow(X,3)
```

Bien:

```
Y=C[1]+X*(C[2]+X*(C[3]+X*C[4]))
```

Diferentes expresiones pueden tener diferentes “cadenas de dependencias”

Optimización de código (MANUAL)

*Operaciones, de mas lenta a más rápida:

-Funciones trascendentales

-raíz cuadrada

-módulo

-división

-multiplicación

-suma/resta/multiplicación por potencia de 2

-división por potencia de dos

-módulo por potencia de dos

Optimización de código (MANUAL) Funciones

-Funciones o subprogramas pueden ralentizar el programa:

1) Salto de una dirección de código a otra
2) Cache de instrucciones se fragmenta si hay demasiadas funciones
3) Cambio de contexto

-No crear funciones innecesarias ?!

-Usar Inlining, optimización interprocedural (si lo permite el compilador).

Optimización de código (MANUAL)

Evaluación de funciones complejas y costosas:

-Si se evaluan muchas veces, se pueden usar tablas +interpolación (ej. Cálculo de Transformada rápida de Fourier).

-Si la tabla no cabe en la caché, puede ser lento.

Optimización de código (MANUAL) Condiciones, Predicción

-En las “pipelines”, las instrucciones son “traídas” (fetched) a la CPU y decodificadas antes de su ejecución.

-Si hay varias opciones (if..elseif..else, switch o select) se predice cual es la más probable; si la predicción falla, se produce un gasto de ciclos (branch misprediction penalty)

-Un bucle “For” también tiene “condición” y “ramas”; decidir, tras cada iteración, si repetir o salir;

Optimización de código (MANUAL) Bucles

La eficiencia de los bucles depende de muchos factores;

-predicción correcta

-acceso a memoria dentro del bucle,

-minimizar el número de veces que se ejecuta

-En procesadores modernos, vectorización

Optimización de código (MANUAL) Bucles

Vectorización: En cada core de los procesadores modernos, hay registros vectoriales e instrucciones vectoriales (SSE (128 bits), AVX (256, 512 bits)).

Dado un bucle “sencillo” sin dependencias de datos, el compilador puede paralelizarlo. En la actualidad la vectorización es clave para rendimiento de bucles

Optimización de código (MANUAL)
Bucles

Los bucles con muy pocas iteraciones no son rentables:

Mal:

```
For (i=0;i<3;i++)
 A[i]=0;
```

Bien:

```
A[0]=0;
A[1]=0;
A[2]=0;
```

Optimización de código (MANUAL)
Bucles

Cuando se anidan varios bucles, el de mas iteraciones debería ser el mas interno (si el problema lo permite)

Mal:

```
for (i=0;i<100;i++)
  for (j=0;j<10;i++)
    for (k=0;k<2;i++)
                {...
                    }
```

Bien:

```
for (k=0;k<2;i++)
      for (j=0;j<10;i++)
            for (i=0;i<100;i++)
                  {...
                     }
```

(Se minimizan el número de inicializaciones y terminaciones de bucle, y puede mejorar el efecto de la vectorización)

Optimización de código (MANUAL) Bucles

Las llamadas a subprogramas dentro de un bucle inhiben las posibilidades de optimización, por parte del compilador

Mal:

```
for (i=0;i<N;i++)
    {  ...
    X=pow(Y,5); }
```

Bien:

```
for (i=0;i<N;i++)
    {...
   X=Y*Y*Y*Y*Y;}
```

Optimización de código (MANUAL) Bucles

Posibilidades para eliminar llamadas de subprograma dentro de un bucle:

- Si el bucle es pequeño y el subprograma grande, llevar el bucle al subprograma.
- Si el Subprograma es pequeño y el bucle es grande, “expandir” el subprograma dentro del bucle: INLINING

Mal:

```
for(i=0;i<N;i++)
    {
     ...
     X=exp(Y,5);
    }
```

Bien:

```
for(i=0;i<N;i++)
    {
     ...
     X=Y*Y*Y*Y*Y;
    }
```

Optimización de código (MANUAL) Bucles

LOOP UNROLLING:

“Desenrollar bucles”

Mal:

```
for(i=0;i<N;i++)
    { …
     A[i]=B[i]*C[i];
    }
```

Bien:

```
for(i=0;i<N;i=i+2)
    { …
      A[i-1]=B[i-1]*C[i-1];
      A[i]=B[i]*C[i];
    }
```

Técnica tradicional de optimización para mejorar el acceso a memoria y minimizar número de iteraciones. La vectorización se puede considerar una forma extrema de unrolling

Hay que evitar que el desenrollamiento perjudique la vectorización

Optimización de código (MANUAL)
Bucles
LOOP UNROLLING: Otro tipo

Mal:

```
for(i=0;i<N;i++)
     { ...
   if (mod(i,2)==0)
       funca(i);
  else
    call funcb(i);

  call funcc(i);
      }
```

Bien:

```
for(i=0;i<N;i=i+2)
     { ...
       funca(i);
       funcc(i);
       funcb(i+1);
       funcc(i+1);
     }
```

Optimización de código (MANUAL) Bucles

LOOP UNROLLING: Puntos de vista

1)Eliminar “Ifs”, mejorar predicción(pipelining).
2)Mejorar acceso a memoria
3)Minimizar número de iteraciones

Técnica un poco desfasada a causa de la vectorización.

Optimización de código (MANUAL) Bucles

"Colapsar Bucles": Convertir un conjunto de bucles anidados en uno sólo.

Mal:

```
Double precision A(4,8,50),B(4,8,50), &
C(4,8,50)
DO k=1,50
     DO j=1,8
          DO I=1,4
            A(i,j,k)=A(i,j,k)+B(i,j,k)*C(i,j,k)
           END DO
     END DO
END DO
```

Bien!?:

```
Double precision A(4,8,50),B(4,8,50), &
C(4,8,50)
DO k=1,1600
          A(k,1,1)=A(k,1,1)+B(k,1,1)*C(k,1,1)
END DO
```

Optimización de código (MANUAL)
Bucles

Combinar Bucles Similares: Convertir un conjunto de bucles similares en uno sólo.

Mal:

```
for(i=0;i<100;i++)
      A[i]=B[i]*C[i]+4;

for(i=0;i<100;i++)
      D[i]=E[i]+5;
```

Bien:

```
for(i=0;i<100;i++)
     {A[i]=B[i]*C[i]+4;
       D[i]=E[i]+5;
     }
```

Atención; es posible que el combinar bucles “desborde” la memoria cache, haciendo el bucle mas “ineficiente” que sin combinar.

Optimización de código (MANUAL)
Bucles

Variables de control del bucle:

No deberían:

- Ser variables globales
- Ser argumentos de llamadas a subprogramas en el bucle.

Si es posible, deberían sacarse los IF de los bucles.

Optimización de código (MANUAL) Estructuras, módulos, clases

Pros:

-Variables que se usan juntas se almacenan juntas

Contras: Es necesario tener cuidado con la “alineación” de los datos

Ejemplo: almacenamiento de imágenes con tripletes (r,g,b)

Optimización de código (MANUAL)

Que hace el compilador para optimizar?

- Inlining de Funciones
- Cálculo y propagación de constantes.
- Eliminación de subexpresiones
- Juntar “ramas” idénticas, eliminar “ramas”.
- Evitar saltos, copiando el código al que se salta.
- Desenrrollamiento de bucles
- Reordenar instrucciones para romper cadenas de dependencias, buscar paralelismo

Optimización de código (MANUAL)

Obstáculos para optimización

-Optimización entre módulos

-Acceso a variables por punteros

-Largas cadenas de dependencia

Optimización de código (MANUAL)

Toma de tiempos

Normalmente, las rutinas de toma de tiempos devuelven una precisión máxima de centésimas de segundo.

En www.agner.org hay funciones para contar ciclos de CPU; Ver manual optimización C++, capítulo 16.