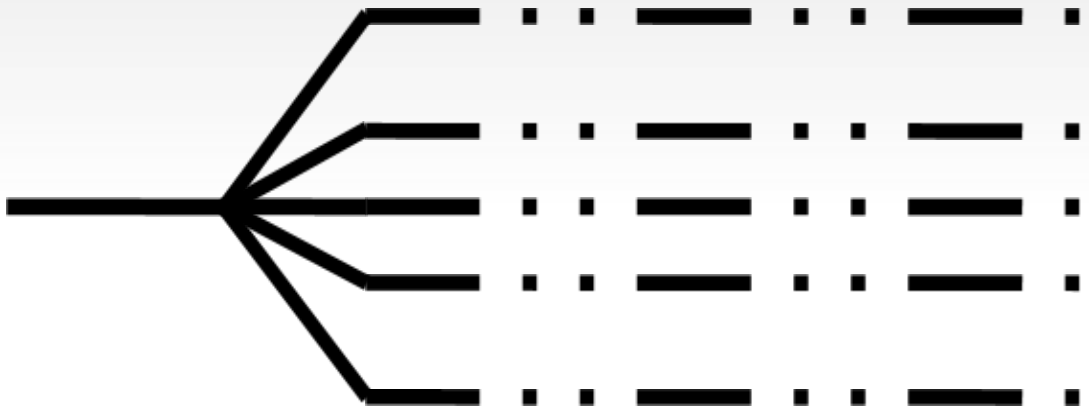
Paralelización automática en GCC



Fernando Álvarez Calleja
Procesamiento Paralelo
2010

¿Qué es GCC?

- Conjunto de compiladores del proyecto GNU.

- Iniciado por Richard Stallman en 1985.

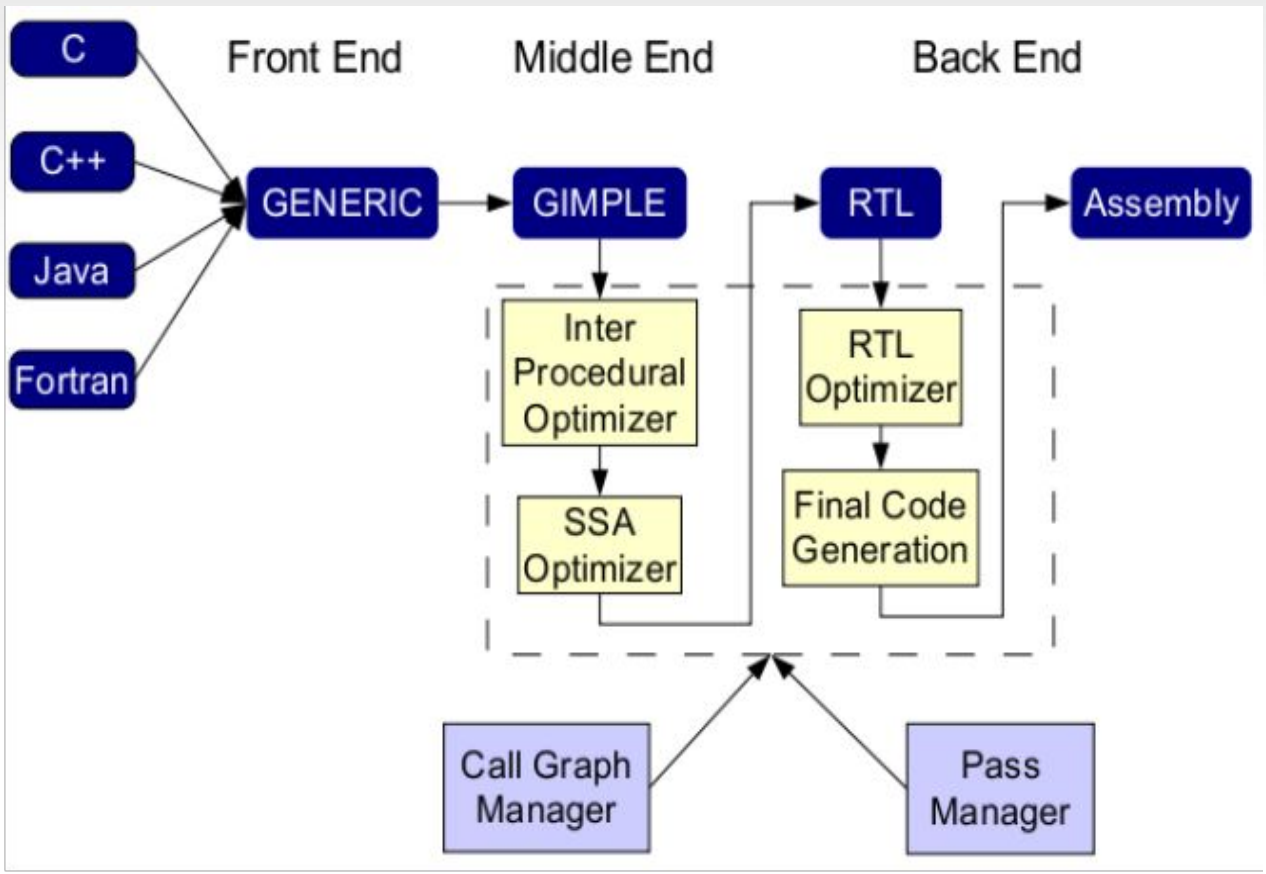
- Compila programas en C, C++, Objetive C, Fortran, así como otros, a lenguaje máquina.

- Inicialmente concebido para máquinas con procesadores de un núcleo.

- Disponible en gran cantidad de arquitecturas y plataformas.

¿Qué es GCC?

ARQUITECTURA



Motivaciones de la paralelización automática

- Aparición de procesadores multicore.

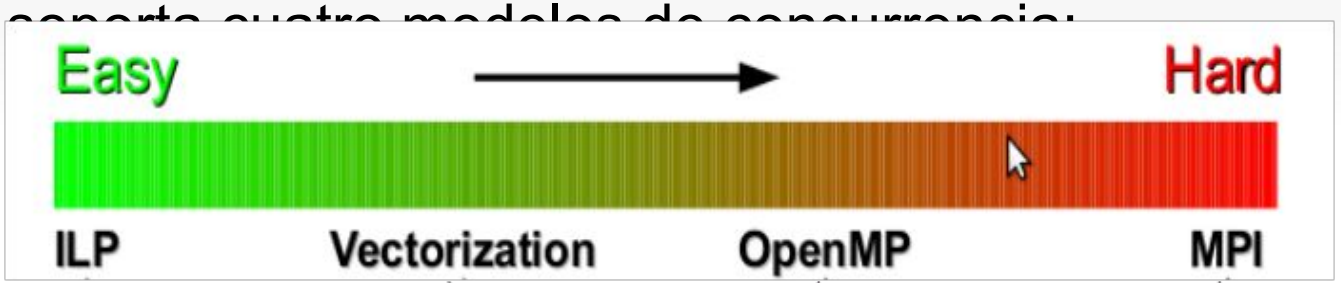
- Programas paralelizables, gran cantidad de bucles.

- Gran ganancia de rendimiento y productividad.

Autopar y GCC

- Autopar: Automatic Parallelization (Generar código paralelizable)

- GCC soporta cuatro modelos de concurrencia:



ILP	Vectorization	OpenMP	MPI
- automático - sin control de usuario - no intrusivo	- automático - opción de compilación - no intrusivo	- manual - directivas de compilación - algo intrusivo	- manual - librerias especiales - muy intrusivo

OpenMP vs MPI

	Pros	*Contras*
OpenMP	-Más fácil de programar y depurar. -Se pueden añadir directivas incrementalmente, paralelización graduada. -Puede ejecutar programas como código en serie. -Declaraciones de código en serie no necesitan modificaciones. -Código más fácil de entender y más fácil de mantener.	-Sólo puede ser ejecutado en máquinas con memoria compartida. -Requiere un compilador que soporte OpenMP. -Mayoritariamente usado para paralelización de bucles.
MPI	-Se ejecuta tanto en memoria compartida como distribuida. -Puede resolver un rango más amplio de problemas que OpenMP. -Cada proceso tiene sus propias variables locales. -Máquinas de memoria distribuida son más caras que las de grande memoria compartida.	-Requiere más cambios de programación el convertir código en serie a paralelo. -Más difícil de depurar. -Rendimiento está limitado por la comunicación de red entre los nodos.

GOMP (GNU OpenMP)

- Implementación de OpenMP para el proyecto GNU.

- En 2007 se incluye las especificaciones de OpenMP v2.5 en GCC v4.2.

- En la versión de GCC 4.3, año 2008, se incluye GOMP implementando la v3.0 de OpenMp.

- Los compiladores C, C++ y Fortran soportan esta implementación de la interfaz OpenMP.

- Disponible en todas las plataformas que soporten hilos POSIX.

- Utiliza la libreria libgomp, que implementa el actual mecanismo para crear hilos, la sincronización y la compartición de datos.

- Dirigido a la generación de código para las transformaciones de auto-paralelización.

GOMP (GNU OpenMP)

-Implementación centrada en cuatro componentes:

1 – Parser (Front-End).

2 – Representación intermedia.

3 – Generación de código (Back-End)

4 – Librería en tiempo de ejecución (libgomp).

GOMP (GNU OpenMP)

Parser
(Front-End)

-Pragmas: **Directivas** (especifica paralelismo y compartición de trabajo) y **Claúsulas** (especifica las propiedasdes de la compartición de datos y la planificación de los hilos).
-OpenMP define pragmas para C, C++ y Fortran: se requieren tres implementaciones, una para cada front end.

-Cada comando OpenMP empieza por #pragma omp, y se reconoce el código en cada uno de los front end a través de cada pragma de procesamiento de código:
C: c-parser.c:c_parser_omp_*
C++: cp/parser.c:cp_parser_omp_*
Fortran: fortran/parse.c:parse_omp_*

-Una vez reconocido los parsers generan la representación GENERIC.
-El análisis de la estructura se realiza durante la conversión a GIMPLE: gimplify.c:gimplify_omp_*andgimplify.c:omp_*.
-Otros análisis como el anidamiento de directivas se realiza después de que la respresentación esté en la forma GIMPLE: omp-low.c:diagnose_omp_structured_block_errors.

GOMP (GNU OpenMP)

Representación Intermedia

- Usado para representar OpenMP extendiéndolo a GENERIC y GIMPLE.

- La estrategia de generación de código es perfilar las regiones paralelas del código en funciones que serán usadas como argumentos en la creación de hilos de la librería libgomp.

- Un código con directivas OpenMP puede tener una expansión GIMPLE que puede ser alta (High GIMPLE) o baja (Low GIMPLE). Este código GIMPLE será ya paralelizable.

GOMP (GNU OpenMP)

Generación de código (Back-End)

- Los nuevos códigos GENERIC y GIMPLE pueden ser usados para la auto-paralelización.

-Para la generación de código, compartición de datos semánticos o sincronización se puede recurrir a las directivas OMP o directamente a las rutinas de libgomp.

- Una vez que código GIMPLE se ha generado, pass_expand_omp puede ser usado para la expansión a bajo nivel y fijar una nueva función en el gráfico de llamadas.

GOMP (GNU OpenMP)

Librería en tiempo de ejecución (libgomp)

- Básicamente es una capa sobre la librería de hilos POSIX.

- Creación de hilos:
- GOMP_parallel_start.
- GOMP_parallel_end.

- Sincronización:
- Servicios directamente mapeados a los de POSIX, excepto
- omp master: Bloquea un hilo con un id diferente de 0.
- omp single: Con o sin la claúsula copyprivate. Puede bloquear todos los hilos excepto uno. GOMP_single_copy_start.

- Trabajo compartido:
-omp for: GOMP_loop_*_start (inicializa el límite del bucle), GOMP_loop_*_next (realiza una iteración en el tiempo para continuar, GOMP_loop_*_end (finaliza el bucle paralelo).
-omp sections: GOMP_sections_start (configura la construcción del trabajo común y guarda el número de secciones encontradas en el cuerpo).
GOMP_sections_next (devuelve la siguiente sección a ejecutar).

GOMP (GNU OpenMP)

GIMPLE bajo

```
foo ()
{
  #pragma omp for
  for (i = 0; i <= 8; i = i + 1)
  do_work (i);
  OMP_CONTINUE
  OMP_RETURN
  return;
}
```

Expansión correspondiente

```
foo ()
{
    /* Lines 3-14 compute the iteration space for
       each thread.  */
 3  D.1330 = __builtin_omp_get_num_threads ();
 4  D.1331 = (unsigned int) D.1330;
 5  D.1332 = __builtin_omp_get_thread_num ();
 6  D.1333 = (unsigned int) D.1332;
 7  D.1334 = 9 / D.1331;
 8  D.1335 = D.1334 * D.1331;
 9  D.1336 = D.1335 != 9;
10  D.1337 = D.1334 + D.1336;
11  D.1338 = D.1337 * D.1333;
12  D.1339 = D.1338 + D.1337;
13  D.1340 = MIN_EXPR <D.1339, 9>;
14  if (D.1338 >= D.1340) goto <L3>; else goto <L0>;
    /* Lines 20-25 compute the first and last value of
       'i' taking the loop increment value into
       consideration.  */
17  # BLOCK 1
19 <L0>:;
20  D.1341 = (int) D.1338;
21  D.1342 = D.1341 * 1;
22  i = D.1342 + 0;
23  D.1343 = (int) D.1340;
24  D.1344 = D.1343 * 1;
25  D.1345 = D.1344 + 0;
    /* Lines 31-34 are the actual loop.  */
28  # BLOCK 2
30 <L1>:;
31  do_work (i);
32  i = i + 1;
33  D.1346 = i < D.1345;
34  if (D.1346) goto <L1>; else goto <L3>;
    /* This barrier is emitted because the loop
       was not marked with the 'nowait' clause.  */
37  # BLOCK 3
39 <L3>:;
40  __builtin_GOMP_barrier ();
41  return;
}
```

Tratamiento de Bucles

Hemos visto cómo se crea código auto-paralelizable, ¿ahora qué?

- Programas con gran cantidad de bucles que hay que manejar y transformar.

- El compilador necesita de un Framework que realize esta acción y se complemente con el Autopar.

Tratamiento de Bucles

- **Framework Lambda**:

  - Permite la transformación de bucles usando matrices no singulares.
  - Podemos hacer transformaciones de sesgo, ampliación, intercambio o inversión.
  - Requiere que la anidación del bucle sea convertida a una forma interna que puede ser transformada a una matriz fácilmente. lambda_loopnest, lambda_loopnest_transform.
  - Hay bucles que no puede manejar: bucles triangulares, bucles con condición if.

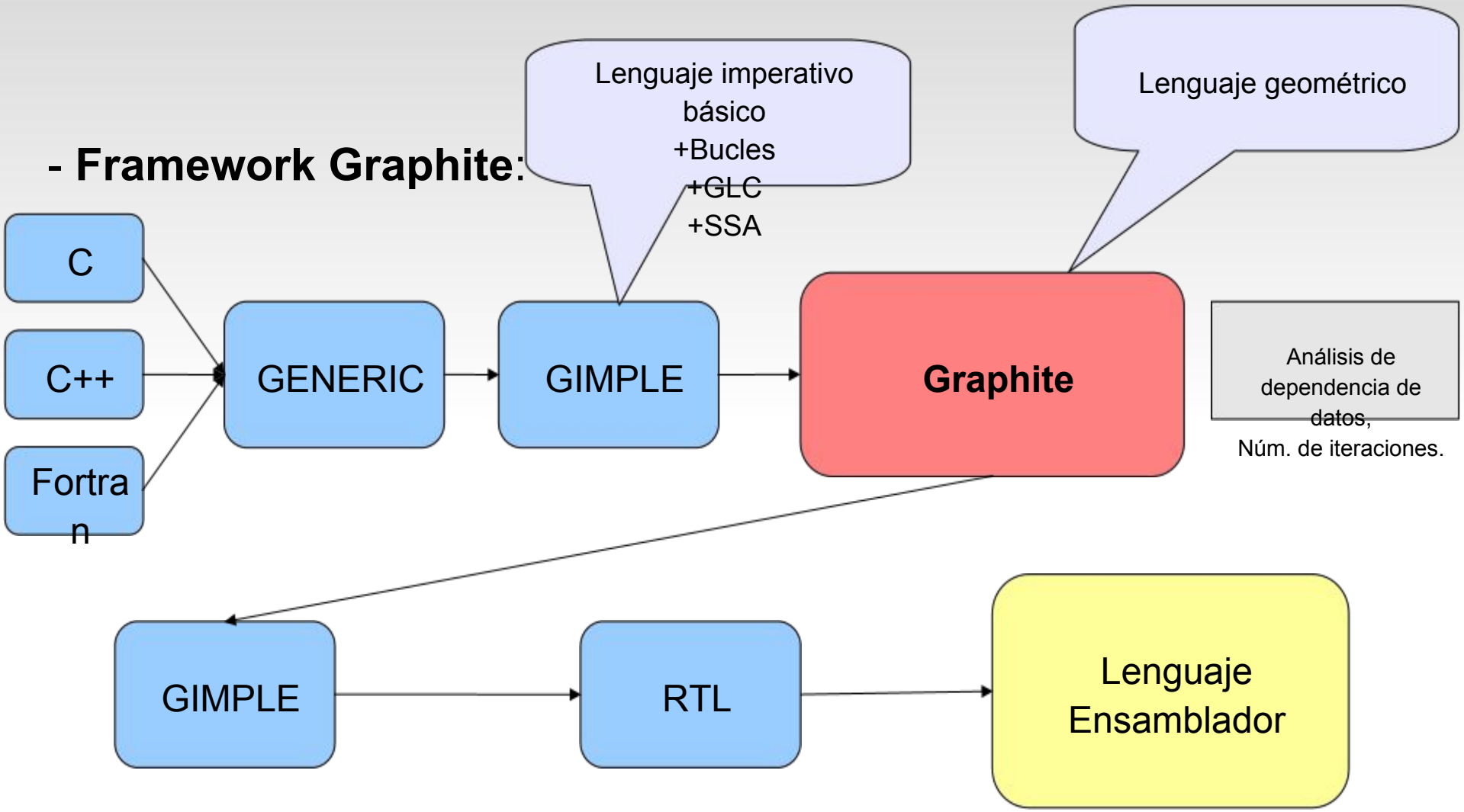
Tratamiento de Bucles

- **Framework Graphite**:

- Gimple Represented as Polyhedra with Interchangeable Envelopes.
- Optimización de bucles de alto nivel.
- Utiliza el modelo poliédrico para paralelizar los bucles.
- Realiza un completo análisis de los bucles anidados y de los métodos de transformación.
- Integrado en la versión 4.4 de GCC en 2009
- Se aplica a todos los lenguajes soportados por GCC.
- Objetivo: integrarlo incrementalmente con autopar.

Tratamiento de Bucles

- Framework Graphite:

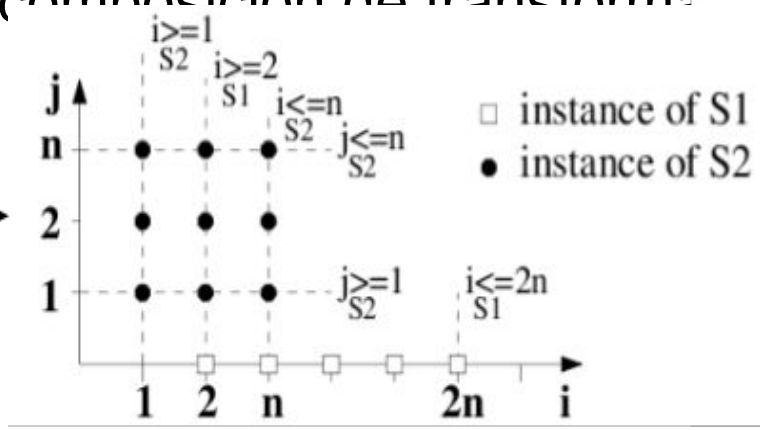


Modelo Poliédrico

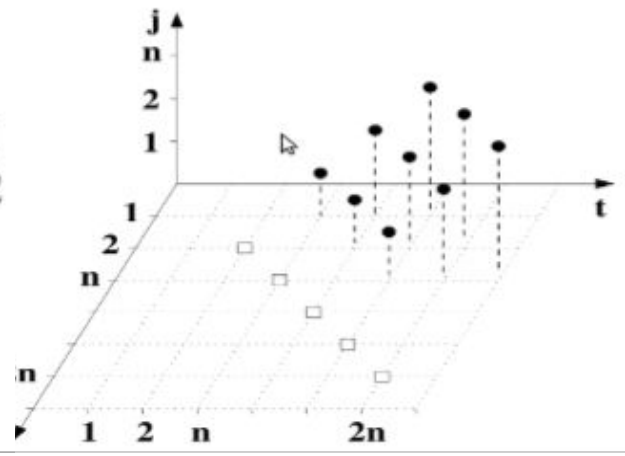
- Motivado por la composición de transformaciones complejas que otros modelos no ofrecen.
- Más motivaciones: la semántica operacional de bucles produce código demasiado largo y un redundante y caótico espacio de busqueda.
- Permite la paralelización total por constantes simbólicas.
- Expresa toda composición de transform

```
      for (i=2; i<=2*n; i++)
S1    |  Z[i] = 0;
      for (i=1; i<=n; i++)
      |  for (j=1; j<=n; j++)
S2    |  |  Z[i+j] += X[i] * Y[j];
```

Forma sintáctica



Dominio poliédrico



Poliedro transformado

Graphite y Autopar

- Enseñar a Graphite que se necesita producir código paralelo: Reconocerá bucles paralelos sencillos usando detección ScoP (*Static Control Parts*) y haciendo un análisis de dependencias, para tener información de si los bucles son paralelizables.

- Dos posibilidades de generación de código:
  - **Graphite** toma nota de los bucles paralelos y pasa esa información a través de ***CLooG*** (librería para la generación de código escaneando un poliedro de tipo Z) para que posteriormente **GOMP** produzca la paralelización.
  - **Graphite** los bucles paralelos y la propia libreria **ClooG** generará el código paralelo.

- Generación de código con ClooG:
  - *cloog_program_generate*
  - *cloog_clast_create*

Probando Autopar

- Se realizarán un para comprobar el rendimiento de la autoparalelización.
  - Tiempo de ejecución de código c no paralelo.
    Opción gcc: O2 for ordinary non-parallel code.
  - Tiempo de ejecución de código c con directiva OpenMP insertada.
    Opción gcc: O2 -fopenmp for openmp directive inserted c code
  - Tiempo de ejecución de código Graphite paralelo c.
    Opción gcc: -O2 -fgraphite-force-parallel
    -ftree-parallelize-loops=$THREADfor Graphite

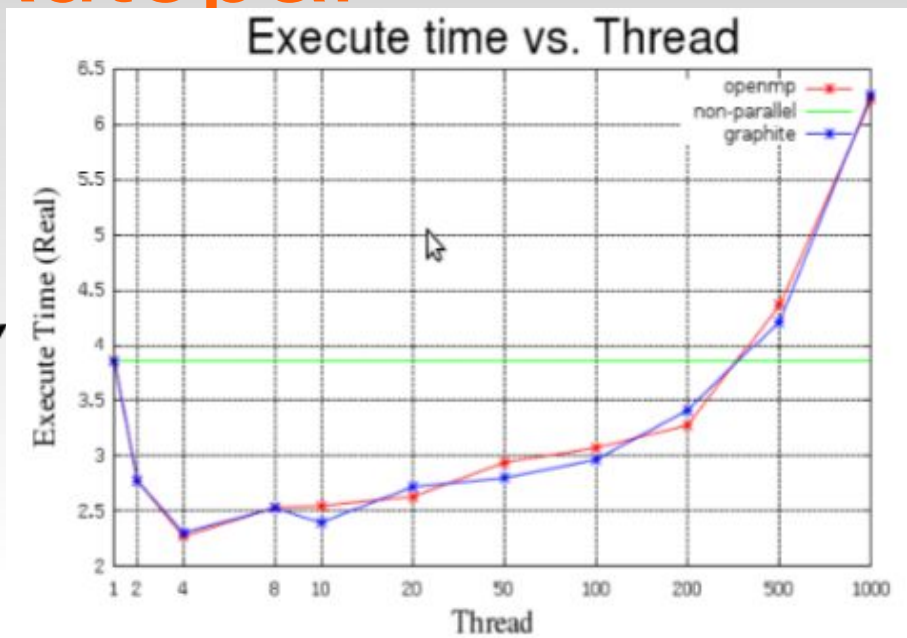
Probando Autopar

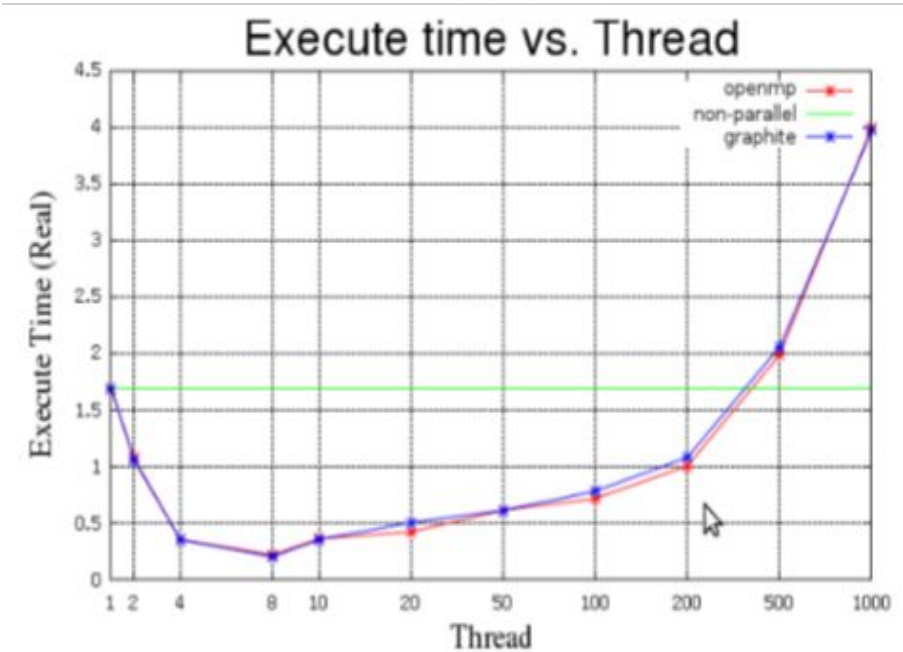
- Test con un bucle anidado

```
#include <stdlib.h>
void parloop (int N)
{
int i, k;
int x[1000000];
for (k = 0; k < 1000; k++){
#pragma omp parallel for shared(N,x) private(i) num_threads (2000)
for (i = 0; i < N; i++)
x[i] = 2*(i + 3);
//Parte no paralela
for (i = 0; i < N; i++)
{
if (x[i] != 2*(i + 3)){
abort();
}
}
}
}
int main(void)
{
parloop(1000000);
return 0;
}
```

Linea #pragma... solo se ejecutará con la opción de openMP

Cuando los threads aumentan, el tiempo de ejecución disminuye. Si hay muchos threads el tiempo aumenta (sobrecarga)



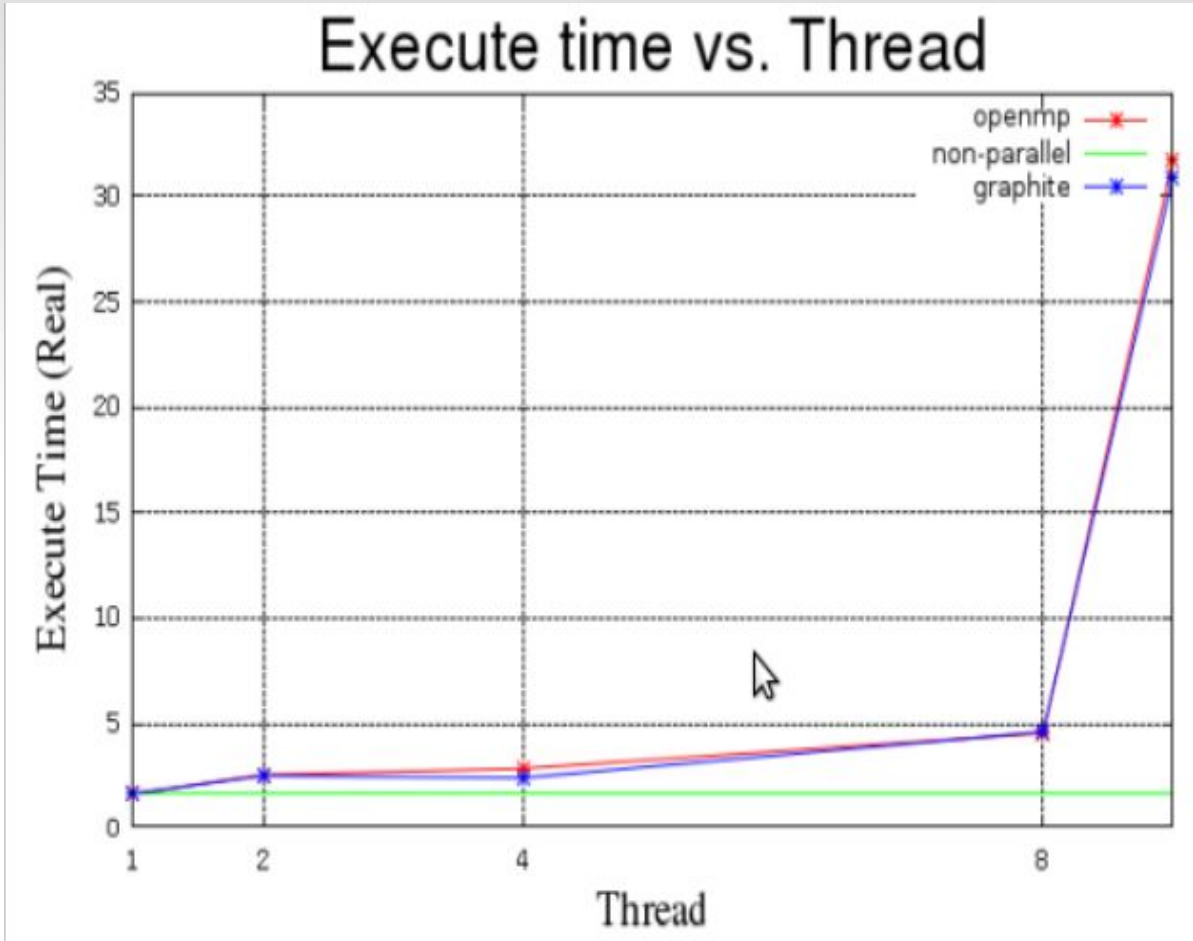


Probando Autopar

- Test con dos bucle anidados

```
#include <stdlib.h>
#define N 1000

int x[N][N];
int main(void)
{
int i, j, k;
for (k = 0; k < 1000; k++)
{
for (i = 0; i < N; i++)
#pragma omp parallel for private(j) num_threads(2)
for (j = 0; j < N; j++)
x[i][j] = i + j + 3;
}
for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
if (x[i][j] != i + j + 3)
abort ();
return 0;
}
```



Con el bucle más interno, el tiempo de ejecución empeora tras la paralelización. Causado por el pequeño número de directivas a ejecutar que están limitadas por la memoria.

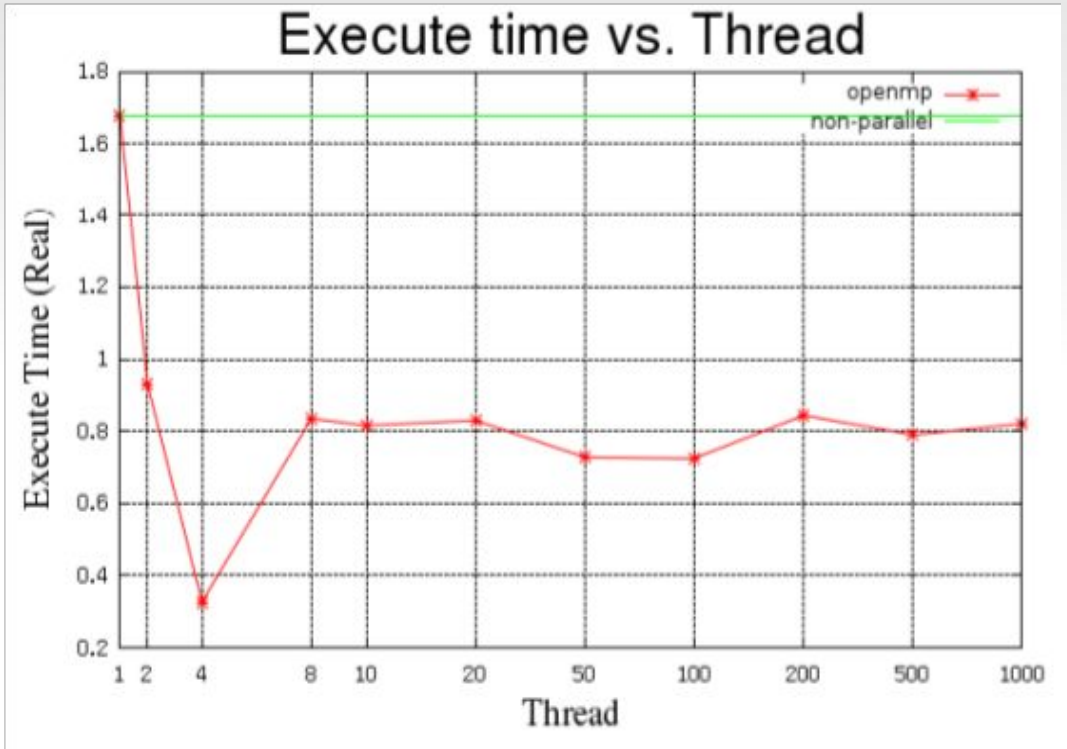
Probando Autopar

- Test con el bucle exterior

- Ya que Graphite no maneja el bucle exterior, probaremos sólo con OpenMP.

```
#pragma omp parallel for private(k) num_threads(2)

for (k = 0; k < 1000; k++)
{
for (i = 0; i < N; i++)
```



Incluso cuando el número de hilos aumenta, se mantiene un buen rendimiento

Referencias

- GCC, web, wiki y mailing list: http://gcc.gnu.org/

-”OpenMP and automatic parallelization in GCC” por Diego Novillo

-”Parallel Programming and Optimization with GCC” por Diego Novillo

-”GRAPHITE: Gimple Represented as Polyhedra” por Sebastian Pop

-”Loop Nest Optimizer of GCC” by Sebastian Pop

-”The GNU OpenMP Implementation” by Free Software Foundation

-”Parallelization and Vectorization in GCC 4.5.0” por GCC Resource Center.

Preguntas, comentarios...