PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC3524 - Tópicos avanzados de sistemas distribuidos

Chapel

Propuesta investigación HPC

4 de julio de 2017

Nicolás Gebauer - 13634941

@negebauer - repo Investigación

Problema

Cada día se tienen más datos disponibles con los cuales uno puede realizar cómputos interesantes. Cada día se tiene computadores más poderosos y juntando varios de ellos se tienen *clusters*. En este contexto se hace importante poder aprovechar estas plataformas *HPC*.
Hoy esto es posible con lenguajes como *C* y *frameworks* adicionales como *OpenMP* y *MPI* que permiten aprovechar los recursos disponibles para hacer cómputos paralelos.

OpenMP permite trabajar en distintas tarea paralelas, aprovechando las únidades de cómputo que tiene disponible la máquina que se utiliza.
MPI permite además de trabajar usando los varios *cores* disponibles en una máquina utilizar distintos nodos, es decir, realizar cómputos en varios computadores al mismo tiempo de manera paralela. Esto permite, por ejemplo, trabajar con grandes *sets* de datos para obtener resultados.
Otras herramientas como *CUDA* y *OpenCl* permiten usar más recursos de los sistemas, como sus tarjetas *GPU* para acelerar los cómputos.

Ante tantas herramientas, el campo del desarrollo de programas *HPC* tiene una barrera de entrada muy grande. Primero se necesita ser compentente en lenguajes como *C* que no son lenguajes modernos, es decir, no tienen una alta prioridad en el aprendizaje de los programadoresde hoy. Luego uno debe aprender sobre las distintas herramientas para poder aprovecharlas.

Solución

Para poder atacar esta problemática la empresa Cray decidió crear un nuevo lenguaje, llamado *Chapel*. El objetivo principal de *Chapel* es acercar la programación orientada a *HPC* a más desarrolladores. Algunos de los objetivos de *Chapel* que es importante destacar.

- Un única herramienta para *HPC*
- Lenguaje alto y bajo nivel
- Lenguaje moderno

El primer punto hace referencia a que el lenguaje engloba muchas formas de hacer trabajos paralelos en varios nodos que actualmente requieren distintas herramientas. Esto facilita el desarrollo ya que solo es necesario aprender a utilizar un nuevo lenguaje.

El segundo destaca la importancia de que el lenguaje facilita el comenzar a realizar trabajos en paralelos a través de directivas de alto nivel. Esto reduce la barrera de entrada para quienes son nuevos en *HPC*. A la vez permite un manejo a bajo nivel, lo que le permite a quienes tienen conocimientos más avanzados poder trabajar más cercano a la máquina, por ejemplo, optimizando de mejor forma el uso de memoria.

Como ejemplo, en *Chapel* correr un proceso en varios nodos es tan simple como ejecutar:

```
// Un task para cada local (nodo) disponible
coforall loc in Locales {
  // Movemos el computo al local
  on loc {
    // Imprimimos un mensaje
    writeln('Hola desde local ', here.id);
      // here hace referencia al local que esta corriendo el codigo
  }
}
```

El tercer punto se refiere a que el lenguaje fue escrito tomando inspiración de lenguajes más moderno como *Python* y *Java*, lo que hace que nuevos desarrolladores se sientan más cómodos al interactuar con el. También tomaron inspiración de *C* para que desarrolladores veteranos no queden excluidos. Por ejemplo, se puede programar con *OOP* si uno lo desea.

Desarrollo

El objetivo de este trabajo es, en pocas palabras, hacer una prueba del lenguaje *Chapel*. Para ello se aprovecha la Tarea 2 que se realizó en el curso con *MPI*. El código resultante puede verse en el siguiente repositorio de github, Junto con instrucciones sobre como activar chapel, configurar algunas variables de entorno (como los *hosts* a utilizar), el comando `chpl` para compilar programas de chapel y como correrlos.

Para probar *Chapel* se desarrollaron dos versiones trabajos. La primera tuvo como objetivo ser una copia y adaptación del código utilizado en la Tarea 2. La segunda versión tuvo un desarrollo más enfocado a aprovechar de mejor manera las ventajas de *Chapel* y lograr mejores resultados. Las versiones son *main.chpl* y *main2.chpl* respectivamente, ambas disponibles en la carpeta *code* del repositorio mencionado.

main.chpl

Se replicó de manera cercana el código desarrollado en la Tarea 2. Primero se lee del archivo de input, el cual es definido a través de la constante de configuración *input*. Esto permite cambiar el archivo a usar al llamar al código compilado la opción `--input=ruta/a/input.txt`.

```
config const input='./test/t3.txt';

const file = open(input, iomode.r);
const reader = file.reader();
var size: int;
reader.read(size);
```

Luego se definen las mismas clases *WSP* y *Route* que fueron usadas en la tarea. La primera define parámetros del problema (costo más bajo, orden de las ciudades, tamaño). La segunda se utiliza para resolver el costo de una ruta, a través de su método *dfs*.

```
  proc updateCost(wsp: WSP) {
    if cost < wsp.cost {
```

```
      wsp.cost = cost;
      wsp.cities = cities;
    }
  }

  proc dfs(wsp: WSP, costs: [] int): void {
    if visits == wsp.size {
      updateCost(wsp);
      return;
    }
    if cost >= wsp.cost then return;
    for destination in 1..wsp.size-1 {
      if !visited(destination) {
        advance(costs, destination);
        dfs(wsp, costs);
        back(costs);
      }
    }
  }
```

Se puede apreciar aquí la lógica principal del programa. Se realiza una exploración *DFS*, donde se visita cada posible ciudad de destino. Si el costo acumulado de la ruta supera al menor costo encontrado se termina dicha exploración (se tiene una cota). Una vez que se completa una ruta se actualiza el costo global, si el de esta ruta es menor.

Dado que todas las ciudades están conectadas una posible manera de paralelizar la exploración *DFS* es que cada unidad de cómputo tome una ciudad para realizar *DFS* a partir de ella. Para lograr este cómputo paralelo se usó la distribución cíclica que ofrece *Chapel*, la cual reparte las ciudades de destino del primer viaje en distintos *tasks*, que son luego ejecutados por cada unidad de cómputo. Esto se realiza de la siguiente forma.

```
const DestinationSpace = {1..size-1} dmapped Cyclic(startIdx=1);
forall destination in DestinationSpace {
  const costsLocal = costs;
  writeln('task ', here.id, ' destination ', destination);
  const route = new Route();
  route.advance(costsLocal, destination);
  route.dfs(wsp, costsLocal);
  writeln('task ', here.id, ' destination ', destination, ' finished');
}
```

Como se puede observar, la distribución de tareas es muy simple en *Chapel*. El iterador `forall` crea tareas de manera inteligente para los dominios que se le entregan. En este caso particular crea la cantidad de tareas correspondiente al número de nodos por la cantidad de cores disponibles en cada uno. Esto a diferencia de un `coforall`, que crea una tarea para cada iteración del dominio que se le entrega. Se prefiere utilizar `forall` en este caso para que *Chapel* se preocupe de cuantos procesos crear en cada nodo. Importante notar la lína donde se define `costsLocal`, ella permite tener una versión local de los costos entre ciudades en el nodo que ejecuta *DFS*, para evitar tener que ir a pedir cada costo a la memoria compartida. Se implementó de esta forma ya que consultar a la memoria compartida los costos tenia un impacto importante sobre el rendimiento.

Esta versión del código tuvo el siguiente rendimiento, el cual se compara con el tiempo que tomo el código usando *MPI*.

t3.txt	*MPI*	*Chapel*
real	0m1.923s	1m43.229s
user	0m0.244s	0m0.004s
sys	0m0.184s	0m0.016s

Como se puede observar *Chapel* toma mucho más tiempo en resolver el problema. En un orden alrededor de 50 veces más. Los creadores de este lenguaje mencionan que han logrado tener resultados no tan lejanos

de otras alternativas, por lo que este resultado llama mucho la atención. Probablemente pueda deberse a que el código esta orientado a replicar el trabajo echo con *MPI* y no está diseñado eficientemente para *Chapel*.

main2.chpl

Para el segundo código se utilizó el mismo sistema de resolución que en el anterior. Calculando los costos de cada ruta con *DFS*, comparando con el costo más bajo encontrado y actualizandolo de ser necesario. Se realizaron algunas modificaciones importantes para optimizar el uso de la memoria, asegurando que cada local tenga disponibles los datos que necesita, reduciendo el uso de la memoria compartida. También este código hace la distribución de las tareas de manera manual, lo que permite tener un código más eficiente. En particular se destacan las siguientes secciones del código.

```
const file = open(input, iomode.r);
const reader = file.reader();
var sizeGlobal: int;
reader.read(sizeGlobal);
file.close();

class WSP {
  var cost: int;
  var cities: [0..sizeGlobal-1] int;
}

const wspGlobal = new WSP(10000);
```

Se declaran solo algunas variables globales necesarias. En este caso la más importante es `sizeGlobal`, el cual es leído desde el archivo a resolver para poder definir la clase *Route* que se utiliza en la resolución del problema.

```
const destinationsForNode: int = (sizeGlobal-1)/numLocales;
var destinationsNode: [0..numLocales-1] domain(int);
var node = 0;
for destination in 1..sizeGlobal-1 {
  destinationsNode[node] += destination;
  if (node == numLocales-1) {
    node = 0;
  } else {
    node += 1;
  }
}
```

Se mapean los primeros destinos (desde la ciudad 0) a los locales disponibles. Esto permite poder trabajar directamente sobre cada local con el rango de destinos que le corresponde, como se verá más adelante.

```
  proc dfs(costs: [] int, size: int, wsp: WSP, wspGlobal: WSP) {
    if visits == size {
      if cost < wsp.cost {
        wsp.cost = cost;
        wsp.cities = cities;
        if cost < wspGlobal.cost {
          wspGlobal.cost = cost;
          wspGlobal.cities = cities;
        }
      }
      return;
    }
    if cost >= wsp.cost then return;
    for destination in 1..size-1 {
      if !visited(destination) {
        advance(costs, destination);
```

```
        dfs(costs, size, wsp, wspGlobal);
        back(costs);
      }
    }
  }
```

Se puede ver que hay una clara separación entre el costo global y el costo del destino que se está resolviendo. Esto reduce el beneficio de cortar algunas ramas antes, pero también reduce la cantidad de veces que se llama a la memoria compartida. Una posible mejora sería actualizar el costo local cada vez que se termina una ruta, dado que en dicho momento se consulta al costo global.

La parte más importante de esta nueva versión se encuentra entre las línas 88 y 117. Las cuales serán revisada a continuación por partes.

```
coforall loc in Locales {
  on loc {
    // Cargamos la matriz de costos en cada local
    var size = sizeGlobal;
    const wsp = wspGlobal;
    var costs: [0..size-1, 0..size-1] int;
```

Se procede a realizar computos en cada local gracias a la directiva `on loc {}` que mueve el trabajo al local correspondiente. Se declaran en dicho local las variables necesarias a partir de las que están en memoria compartida, para reducir los llamados a la anterior.

```
    const file = open(input, iomode.r);
    const reader = file.reader();
    reader.read(size);
    for i in 0..size-1 {
      for j in i+1..size-1 {
        var cost: int;
        reader.read(cost);
        costs[i, j] = cost;
        costs[j, i] = cost;
      }
    }
    file.close();
```

Cada local realiza el trabajo de leer los costos del archivo de *input*. Esto no es óptimo, pero es la única solución que se pudo encontrar para asegurar que cada local disponga de los costos en su memoria local. La guía de *Chapel* que habla sobre como manejar en que local se guarda cada dato no está terminada aún (ver final de la página).

```
    forall destination in destinationsNode[here.id] {
      writeln('task ', here.id, ' destination ', destination);
      const route = new Route();
      route.advance(costs, destination);
      route.dfs(costs, size, wsp, wspGlobal);
      writeln('task ', here.id, ' destination ', destination, ' finished');
    }
```

Cada local trabaja de manera paralela creando tareas para explorar cada destino, lo que permite aprovechar los recursos de cada únidad de cómputo del local (*CPUs*). Se utiliza el *id* del local para saber que destinos le corresponde explorar.

Esta versión del código tuvo el siguiente rendimiento, el cual se compara con el tiempo que tomo el código usando *MPI*.

t3.txt	*MPI*	*Chapel* `main`	*Chapel* `main2`
real	0m1.923s	1m43.229s	0m9.728s
user	0m0.244s	0m0.004s	0m0.012s
sys	0m0.184s	0m0.016s	0m0.016s

qweqwe

Referencias

Chapel chapter, Bradford L. Chamberlain, *Programming Models for Parallel Computing*, edited by Pavan Balaji, published by MIT Press, November 2015. Disponible en http://chapel.cray.com/publications/PMfPC-Chapel.pdf