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

ATOM BORRÓ TODO LO QUE HABÍA ESCRITO AQUÍ

Se puede observar que Chapel sigue teniendo un rendimiento peor que MPI, en este caso de alrededor de 5 veces. Pero si se compara con el código de Chapel anterior se logro reducir el tiempo en alrededor de 14 veces. Esto se alinea con que el código trabaja más cercano al mismo hardware. Esta es también una de las ventajas destacables de Chapel, permite trabajar de manera sencilla los programación en paralelo, pero también permite una configuración más detallada, lo cual es el caso en el segundo código. Con un manejo más directo aún de las tareas se podría, probablemente, lograr mejorar este tiempo.

Es de destacar el echo de que el código es más corto que el desarrollado con MPI y mucho más legible. También es más cómodo a la hora de escribirlo, dada su cercania con lenguajes más modernos.

En conclusión, Chapel es una herramienta prometedora para trabajar en HPC. Con el equipo que tiene trabajando constantemente en el para mejorar su rendimiento, junto con el aporte de la comunidad, se logrará reducir más la brecha en comparacion a las alternativas lideres actualmente.

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