Práctica 7:
Parallel Machine Scheduling Problem with Dependent Setup Times

26 abril 2021

Gabriel García Jaubert

Universidad de La Laguna

Gabriel García Jaubert
alu0101240374@ull.edu.es
Diseño y Análisis de Algoritmos

Práctica 7: Parallel Machine Scheduling Problem with Dependent Setup Times

Gabriel García Jaubert
alu0101240374@ull.edu.es
Diseño y Análisis de Algoritmos

Arquitectura

El sistema sobre el que se han realizado las pruebas posee un Intel core i5 9600k a una frecuencia de 3,7 Ghz sin overclock y 9 MB de caché. Tiene 6 núcleos y 6 hilos. El punto más importante es que las pruebas se han hecho en una máquina virtual usando VirtualBox. La máquina virtual consta de una memoria RAM de 6 Gb a una velocidad de 3200 Mhz y una memoria de disco de 20 Gb. El programa ha sido desarrollado en C++.

Estructuras de datos más frecuentes

Para el diseño, se han creado numerosas clases para representar cada componente del problema y abstraer los principales conceptos como la máquina o las tareas. La clase "principal" es PMSP (Parallel Machine Scheduling Problem). En esta clase almacenamos las solución(S), que es un vector de máquinas con la mejor combinación en el momento. También guarda toda la información que lee por fichero. Los métodos más interesantes son getZClassic(), que utiliza el algoritmo clásico para obtener los TCT de todas las máquinas y para más tarde sumarlos, o también computeSolution(). Esta última función llama a la función computeSolution() de la clase Strategy, esto puede hacerse gracias a que los objetos de la clase Pmsp poseen un puntero de la clase Strategy que es capaz de llamar a esta función. Dada que esta función computeSolution() es abstracta, depende de a qué tipo de clase apunte nuestro puntero Strategy, podremos darle un comportamiento u otro, esto se llama patrón estrategia. Para la práctica no se han utilizado estructuras complejas, solamente vectores proporcionados por la librerías estándar de C++.

Algoritmos

Los algoritmos principales para esta práctica han sido:

- Greedy: inicializamos todas las máquinas con la tarea más corta partiendo desde la tarea 0. Una vez hecho esto, calcularemos que tarea es la que menos incremento de TCT tiene, probando todas las máquinas, y todas las posiciones.
- myGreedy: en este algoritmo también tenemos una fase inicial de añadir la menor tarea a cada máquina. Después, para rellenar estas máquinas, asumimos que la máquina con menor TCT tendrá el menor incremento, y después, añadimos la tarea con menor tiempo entre el tiempo de preparación desde la última tarea de la máquina y el tiempo de ejecución.
- Grasp: para la fase inicial y la fase de relleno, hacemos igual que greedy, pero en vez de añadir la mejor, la añadimos a una lista restringida de candidatos (LRC) de tamaño K que pasamos como parámetro. Para añadir una tarea simplemente cogemos una aleatoria de esta LRC. Más tarde viene la fase de postprocesamiento. Aquí simplemente buscamos un óptimo local a partir de la solución aleatoria que nos ha dado la fase constructiva. Esto lo hacemos probando según el movimiento que escojamos, todas las tareas en todas las

posiciones, Si hay una mejora seguimos hasta que no haya más, eso significa que hemos llegado a un óptimo local.

- GVNS: para este algoritmo hacemos un bucle que se ejecutará N_ITERACIONES veces donde N_ITERACIONES es un parámetro que indica el usuario. Este bucle comienza con conseguir un punto inicial gracias al algoritmo Grasp. Una vez conseguido haremos Kmax veces: un shake para movernos a un punto de un entorno más amplio que moviendo una sola tarea. Para hacer esto el shake mueve **dos** tareas. Una vez en ese punto, buscaremos un óptimo local con VND, que mezcla una búsqueda con todas las estructuras de entorno que queramos. Si encuentra una mejora tras comprobar todos los puntos del entorno L, ese punto será el nuevo punto de partida, y seguirá buscando mejoras, si no encuentra pasará a comprobar en el entorno L + 1, y así hasta Lmax. Si encuentra una mejora volveremos a empezar desde L = 0. Una vez encontrado un óptimo local con VND, y si es mejor que el óptimo local que existía en memoria lo guardamos como mejor çoptimo local. Una vez hecho esto volvemos al bucle más externo donde inicializamos en un punto aleatorio gracias a grasp, y así sucesivamente hasta N_ITERACIONES veces.

Simular experimento

Para simular el experimento debe escribirse make en la carpeta principal del proyecto. Esto creará un ejecutable en el directorio bin/. Se deberá pasar como parámetro el fichero que se quiere ejecutar. Un ejemplo de ejecución sería el siguiente.

./bin/main ./data/I40j_2m_S1_1.txt > out.txt

Es recomendable enviar la salida a un fichero, como por ejemplo *out* para poder visualizar la salida más cómodamente.
Para variar los parámetros experimentales como K, Kmax, se encuentran en las cabeceras de sus respectivas clases, es decir, K y ITERATION_LIMIT de Grasp se encuentran en Grasp.h, así como Kmax y N_ITER_MAX en gvns.h. Cada vez que se cambie una constante debe escribir *make clean*, y después *make.*

Gabriel García Jaubert
alu0101240374@ull.edu.es
Diseño y Análisis de Algoritmos

Resultados obtenidos

Voraz

Problema	n	Ejecución	TCT	CPU(ms)
2m	40	1	13640	77
2m	40	2	13640	80
4m	40	1	7363	79
4m	40	2	7363	76
6m	40	1	5365	85
6m	40	2	5365	74
8m	40	1	4484	95
8m	40	2	4484	100

Voraz propio

Problema	n	Ejecución	TCT	CPU(ms)
2m	40	1	13739	~0
2m	40	2	13739	~0
4m	40	1	7535	~0
4m	40	2	7535	~0
6m	40	1	5556	1
6m	40	2	5556	1
8m	40	1	4542	1
8m	40	2	4542	1

Gabriel García Jaubert
alu0101240374@ull.edu.es
Diseño y Análisis de Algoritmos

Grasp constructivo

LRC = 2

Problema	n	Ejecución	TCT	CPU(ms)
2m	40	1	13990	26
2m	40	2	13851	26
4m	40	1	7542	29
4m	40	2	7377	28
6m	40	1	5413	31
6m	40	2	5237	32
8m	40	1	4457	34
8m	40	2	4311	34

LRC = 3

Problema	n	Ejecución	TCT	CPU(ms)
2m	40	1	14239	36
2m	40	2	13968	38
4m	40	1	7890	40
4m	40	2	7688	42
6m	40	1	5434	46
6m	40	2	5384	46
8m	40	1	4483	49
8m	40	2	4595	48

Grasp

Para visualizar las tablas para el algoritmo Grasp, visite este enlace:

https://github.com/alu0101240374/p7-TCT/blob/informe/grasp_tables.md

Gabriel García Jaubert
alu0101240374@ull.edu.es
Diseño y Análisis de Algoritmos

GVNS

Los movimientos elegidos para crear los entornos para VND han sido:
1º Intercambio/intra máquina
2º Reinserción/ intra máquina
3º Intercambio/ inter máquina
4º Reinserción/ inter máquina

- Límite iteraciones fijas para Grasp = 3
- Movimiento elegido para crear Grasp = Reinserción entre máquinas
- Para el método shake reinsertamos dos tareas aleatorias en dos máquinas distintas también aleatorias

LRC = 2

ID	n	Ejecución	TCT	CPU(ms)	iteraciones	Kmax
2m	40	1	13193	30491	50	5
4m	40	1	7068	44857	50	5
6m	40	1	5134	68562	50	5
8m	40	1	4203	91099	50	5
2m	40	1	13064	63669	100	5
4m	40	1	7057	99126	100	5
6m	40	1	5115	136633	100	5
8m	40	1	4168	177633	100	5

LRC = 3

ID	n	Ejecución	TCT	CPU(ms)	iteraciones	Kmax
2m	40	1	13193	30491	50	5
4m	40	1	7137	49019	50	5
6m	40	1	5133	67931	50	5
8m	40	1	4194	92758	50	5
2m	40	1	13061	69853	100	5
4m	40	1	7072	91862	100	5
6m	40	1	5106	141474	100	5
8m	40	1	4186	185500	100	5

Gabriel García Jaubert
alu0101240374@ull.edu.es
Diseño y Análisis de Algoritmos

LRC = 2

ID	n	Ejecución	TCT	CPU(ms)	mejora	Kmax
2m	40	1	13123	79817	50	5
4m	40	1	7106	44492	50	5
6m	40	1	5083	87759	50	5
8m	40	1	4190	92061	50	5

LRC = 3

ID	n	Ejecución	TCT	CPU(ms)	mejora	Kmax
2m	40	1	13200	48563	50	5
4m	40	1	7083	86154	50	5
6m	40	1	5082	121795	50	5
8m	40	1	4161	168112	50	5

Conclusión

Para problemas tan complejos como este, los algoritmos han dado resultados mejores a medida que su tiempo es mayor, como es de esperar, pero la calidad de la solución es mucho mayor a poco que el tiempo aumente. Es decir, un algoritmo GVNS es mucho más eficaz que uno grasp, porque aunque tarde más, nos dará una mejor solución.