Universidad de Talca
Facultad de Ingeniería
Ingeniería Civil en Computación

Monte Carlo Básico Distribuido

Sistemas Distribuidos

Felipe A. Román Miranda
froman10@alumnos.utalca.cl

Curicó, 26 de mayo de 2016

Introducción

En el presente trabajo se pretende dar a conocer como se creo un agente básico para jugar ajedrez. Para esto, las decisiones del agente fueron tomadas a través de un análisis Montecarlo Distribuido. Posterior a esto, se evaluará el tiempo total de ejecución variando la cantidad de recursos utilizados así como la eficiencia de cada caso.

Conocimientos Básicos

Análisis Montecarlo

Para un escenario dado, de entre varias decisiones posibles, se espera poder tomar la que entregue mayor ganancia. Montecarlo al ser un algoritmo no determinista, permite simular cada una de los escenarios posibles de manera aleatoria, llegando a un estado final, y obteniendo por cada uno de ellos un resultado posible. Luego se elige que resultado entrego el mayor beneficio para finalmente tomar la decisión ideal.

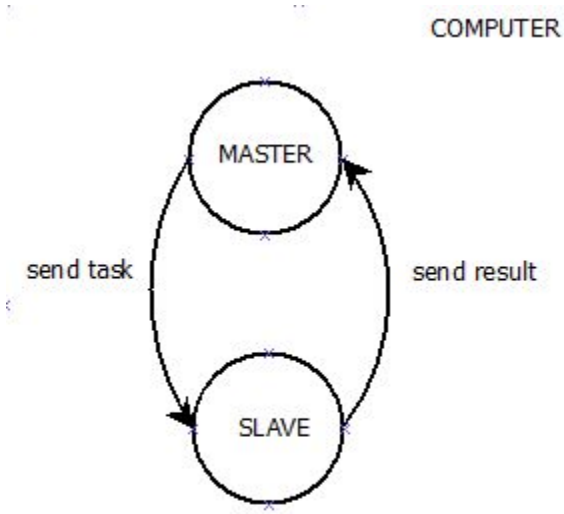
Aplicación Distribuida

Cuando hablamos de una aplicación distribuida, estamos en un escenario donde se intentan utilizar todos los recursos disponibles en una red, para obtener un resultado deseado en el menor tiempo posible. Para esto existen distintos protocolos de comunicación según la tecnología utilizada (por ejemplo el framework rmi para java). Estas aplicaciones pueden tener comunicación síncrona y/o asíncrona según sea la necesidad Además son creadas bajo arquitecturas como Cliente Servidor, modelo tres capas, o multicapa.

Trabajo Realizado

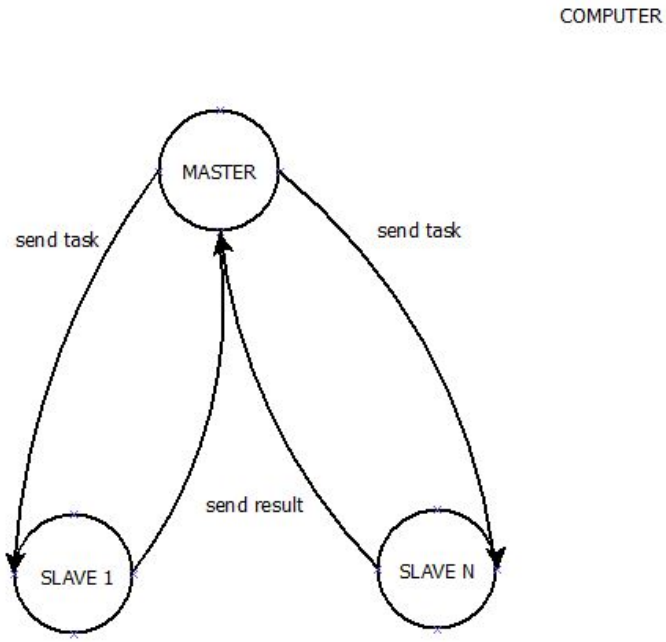
Agente Montecarlo Distribuido Básico para Ajedrez

Un agente representa a un participante de un juego de ajedrez. Dado un estado de tablero, el agente debe realizar la mejor mejor para asegurar su triunfo. Cuando el agente utiliza un análisis Montecarlo, simula aleatoriamente el resultado de cada jugada posible asignándole un valor. Luego de haber analizado cada una de las jugadas, las compara para finalmente tomar una decisión.

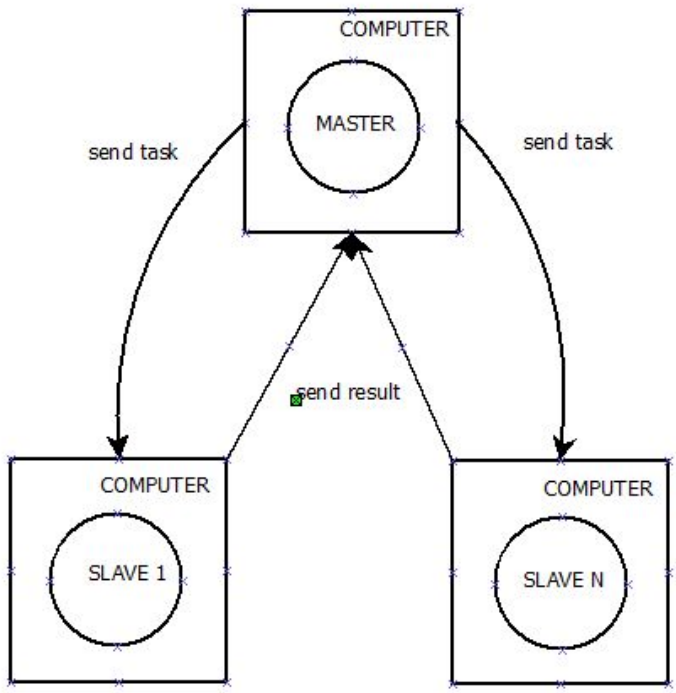


Escenario donde se realiza la simulación con un CPU en el mismo equipo.

Un agente que realiza su análisis Montecarlo de forma distribuida, dispone de varios recursos, tanto núcleos en su propio equipo, como otros dispositivos conectados mediante una red. El agente toma un rol de maestro, el cuál asignará a cada uno de los recursos (esclavos) una jugada para evaluar. Cuando cada esclavo ya tenga terminada su tarea, enviará el resultado de la evaluación al maestro. Por último cuando el maestro tenga todos los resultados, podrá tomar la decisión de qué jugada es la mejor.



Escenario donde se realiza la simulación con N CPUs en el mismo equipo.



Escenario donde se realiza la simulación con N CPUs en distintos equipos.

MonteCarloAgentServer.java

Esta clase es utilizada para para mantener una conexión constante con el maestro, que permita recibir de forma asíncrona el resultado de cada uno de los esclavos. Esta clase además recibe las ips de los esclavos, el número de simulaciones por juego, el número de simulaciones por movimiento posible y el tablero a evaluar.

MonteCarloAgentInterface.java

Esta clase entrega los métodos que serán utilizados para generar la comunicación con el maestro. Define dos método principales. El primero es initGame, quien se encarga de iniciar la simulación completa. El segundo método es receiveResult, el cual permite recibir al maestro recibir cada uno de los resultados calculados por los esclavos.

MonteCarloAgent.java

Esta clase implementa toda la lógica definida por su interfaz. Distribuye la cantidad de movimientos a analizar de forma equitativa entre los esclavos, y toma la decisión en base a los resultados obtenidos. Cuando asigna tareas a cada recurso, lo hace mediante la clase Request a través del uso de threads.

Request.java

Esta clase permite enviar N movimientos para analizar a un esclavo en particular. Recibe el tablero, el número de simulaciones a realizar, la ip del esclavo, desde y hasta que movimiento analizará, y un identificador . El thread se cierra cuando se enviaron todas las peticiones. Esta clase es quien finalmente se comunicará con cada esclavo.

MonteCarloSlaveServer.java

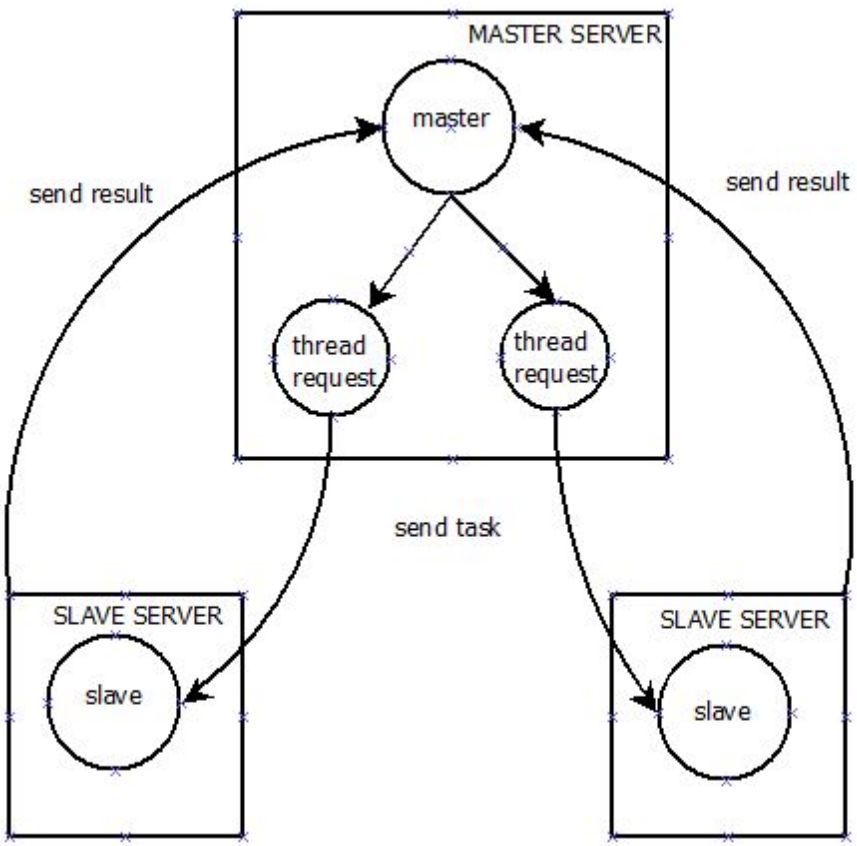
Esta clase es utilizada para para mantener una conexión constante con el esclavo, que permita recibir de forma asíncrona la petición del maestro.

MonteCarloSlaveInterface.java

Esta clase entrega los métodos que serán utilizados para generar la comunicación con el esclavo. Se define el método playOut, mediante el cual se realizará la simulación para cada jugada. Recibe el tablero, el movimiento, el número de simulaciones a realizar, la ip del maestro, y un identificador

MonteCarloSlave.java

Esta clase implementa toda la lógica definida por su interfaz. Realiza N simulaciones aleatorias para un movimiento en particular, asignándoles valores según los resultados (1 si ganas, 0 si empata, -1 si pierde). Finalmente los resultados se acumulan para finalmente ser enviados al maestro.



Esquema que representa la aplicación desarrollada.El maestro mediante request envía tareas a los esclavos, y estos le responden directamente.

Pruebas

Ambiente

Las pruebas fueron realizadas en una máquina virtual con sistema operativo Fedora 23 cuyas características son:

- 3 Gb de RAM
- 4 núcleos
- 20 GB HDD

Además se utilizó imagenes con distribución ubuntu ejecutadas mediante Docker.

El maestro y un esclavo fueron ejecutados en la maquina virtual principal (fedora), mientras que los cuatro esclavos restantes fueron ejecutados en las máquinas virtuales ubuntu.

Número de Esclavos	Tiempo Promedio en mseg	SpeedUp	Eficiencia
1	7019	1	1
2	4898	1,43303389	0,71651695
3	4575	1,53420765	0,51140255
4	4431	1,5840668	0,3960167
5	4616	1,52058059	0,30411612

Pruebas realizadas con 20 ejecuciones del agente, y 30 evaluaciones por cada jugada.

En este gráfico podemos apreciar como el tiempo promedio de ejecución baja drásticamente desde un esclavo a dos. Luego el declive es gradual hasta el cuarto esclavo. A partir del quinto, el tiempo promedio de ejecución comienza a subir, por lo que no se justifica utilizar más recursos. Importante destacar cómo el overhead del tiempo de comunicación va tomando relevancia a medida que aumenta la cantidad de esclavos disponibles.

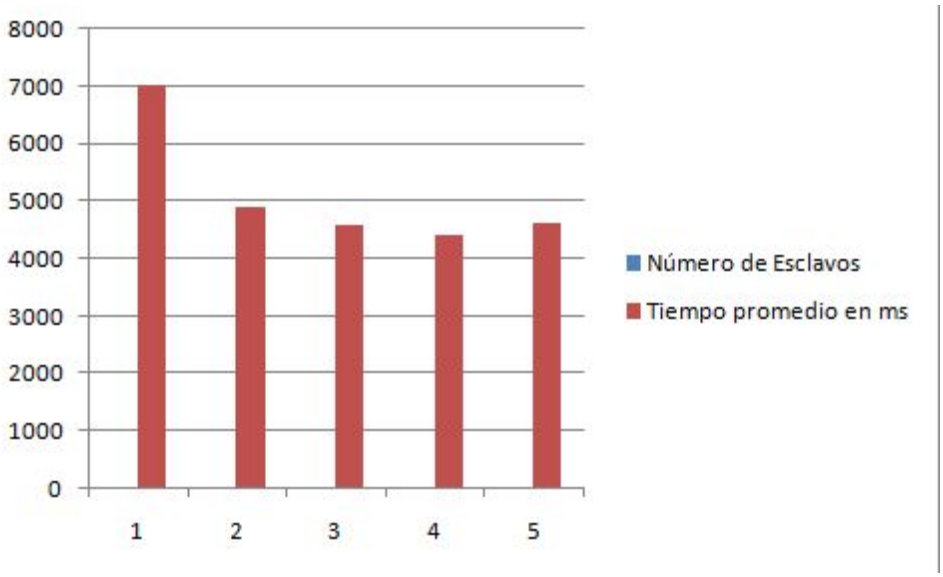


Gráfico de Número de Esclavos vs Tiempo promedio en milisegundos.

En este gráfico podemos apreciar como el SpeedUp aumenta proporcionalmente al aumento de esclavos, sin embargo llegando a cierto punto, este empieza a disminuir debido a la sobrecarga del tiempo de comunicación. Otra razón para que el speedUp comience a bajar puede ser el límite de threads que se pueden crear en el maestro.

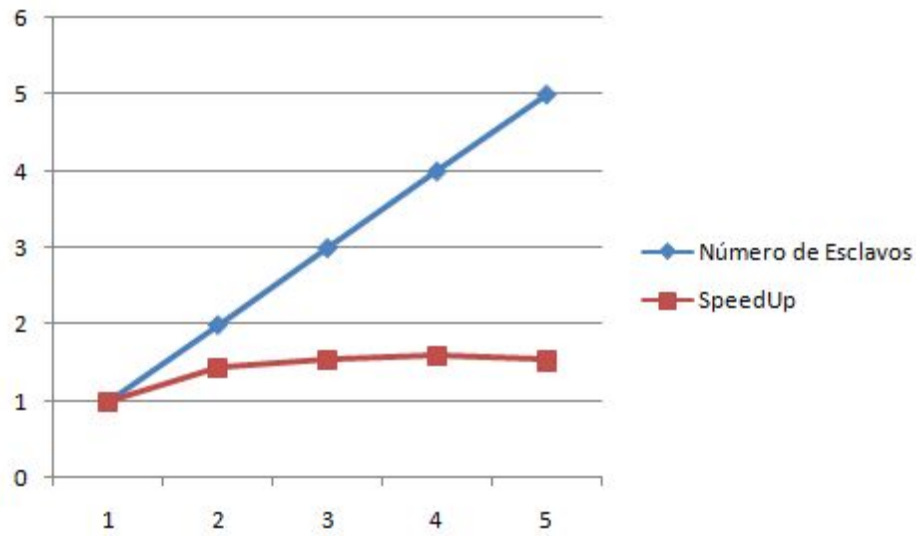


Gráfico de Número de Esclavos vs SeepUp

Finalmente podemos ver como la eficiencia va decayendo a medida que aumenta el número de esclavos. Esto nos dice que el porcentaje de paralelización del programa es cada vez menor. Con dos esclavos, tenemos un 71% de paralelización. Con tres tenemos un 51% mientras que con cuatro tenemos 39%. Finalmente con cinco esclavos tenemos 30%.

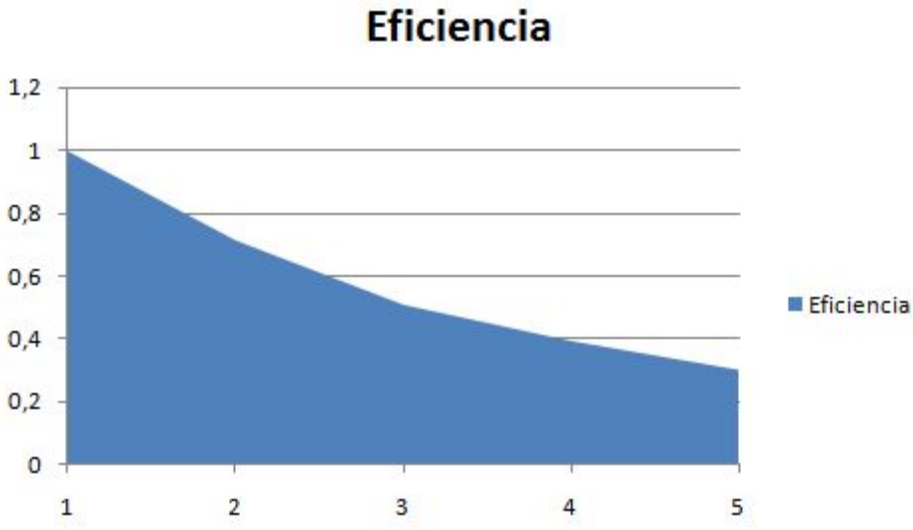


Gráfico de declive de la eficiencia.

Conclusión

Realizar una aplicación distribuida nos permite utilizar los recursos disponibles para aumentar la productividad del programa, obteniendo resultados en menor tiempo. Para esto es importante definir correctamente la distribución de tareas según los recursos disponibles. Otro factor a destacar es como el tiempo de comunicación va tomando relevancia a medida que aumenta el número de recursos, hasta llegar a un punto en que agregar un nuevo participante no es justificable debido a la sobrecarga que genera en el sistema. A esto se agrega la importancia de la calidad de conexión que existe entre el servidor y los cliente.

Otro punto a destacar es la gestión de los recursos. Esto se debe a que la asignación de tareas realizadas en el momento oportuno permite tener trabajando a los esclavos el mayor tiempo posible, aprovechando los recursos al máximo, y permitiendo disminuir los tiempos de cálculo.