II Workshop de Procesamiento Distribuido y Paralelo (II WPDP)

ANÁLISIS COMPARATIVO DE ESQUEMAS DE BALANCEO DE CARGA EN AMBIENTES DE PROCESAMIENTO DISTRIBUIDO

Diego O. Scarpa, Waldemar Baraldi, Gustavo E. Vazquez, Ignacio Ponzoni y Jorge R. Ardenghi

Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Av. Alem 1253 – CP 8000 - Bahía Blanca – ARGENTINA

Resumen

En este trabajo se presentan distintos enfoques para resolver el problema de balanceo de carga sobre ambientes de procesamiento distribuido. Las arquitecturas propuestas están organizadas bajo un filosofía Master-Worker y su implementación fue realizada en C++ utilizando la librería de pasaje de mensajes PVM. Las distintas políticas de balanceo de carga fueron comparadas para una amplia variedad de casos de estudio con el objetivo de estableces los alcances y limitaciones que cada alternativa.

1. INTRODUCCIÓN

En la actualidad el cómputo paralelo sobre ambientes distribuidos constituye una valiosa herramienta para el tratamiento de aplicaciones que requieren procesar un gran volumen de datos en muy poco tiempo. De este modo, problemas costosos de resolver en forma secuencial han logrado beneficiarse con este enfoque (Tinetti y De Giusti, 1998). La idea básica detrás del paralelismo es reducir un problema grande en una secuencia de tareas menores que puedan ejecutarse simultáneamente sobre diferentes procesadores. De este modo, el tiempo de cómputo de un programa ejecutado en una computadora con un único procesador puede reducirse hasta n veces (caso ideal) cuando el esfuerzo es distribuido adecuadamente entre n procesadores.

El procesamiento distribuido ofrece la posibilidad de incrementar el poder de cómputo en diversos problemas, pero al mismo tiempo obliga a desarrollar nuevos algoritmos y técnicas que permitan explotar al máximo las ventajas de esta forma de procesamiento. En tal sentido, un tópico de especial interés dentro de esta disciplina es el problema del balanceo de carga.

Cuando se divide un problema entre un número fijo de procesos que serán ejecutados en paralelo, cada procesador efectuará una cantidad conocida de trabajo. En otras palabras, se asume que las tareas son simplemente distribuidas entre las unidades de procesamiento disponibles, sin considerar los distintos tipos de procesadores involucrados y sus capacidades de cómputo. Sin embargo, esta simplificación de la realidad puede conducir a desperdiciar los recursos disponibles, llegando a veces incluso a soluciones más lentas que sus contrapartidas secuenciales. Por tal motivo, antes de implementar un algoritmo paralelo distribuido es necesario decidir una política adecuada para el balanceo de carga entre los distintos nodos de procesamiento.

Existen varios enfoques para atacar este problema (Chaudhuri, 1992; Chalmer y Tidmus, 1996; Buyya 1999), los cuales se agrupan en dos líneas: balanceo estático, en el cual la distribución de los procesos queda establecida antes de la ejecución del programa, y balanceo dinámico o por demanda, en donde las tareas se van asignando durante la ejecución según el nivel de carga de cada procesador. Cada una de estas políticas tienen sus ventajas y desventajas, y la elección entre una y otra depende del tipo de problema a tratar. En general, si se conoce de antemano el número de tareas a distribuir y se pueden estimar sus tiempos de ejecución, resulta más eficiente usar un balanceo estático. En cambio, cuando no se puede estimar la cantidad de procesos a correr, ni el tiempo de cómputo que consumirán, un enfoque dinámico es mejor. Dentro de los esquemas dinámicos se puede distinguir entre centralizados y descentralizados. Los primeros proponen un manejo centralizado de la distribución de tareas mediante un proceso maestro que se encarga de mantener una carga balanceada entre los distintos procesadores. Esta política, posee la desventaja de congestionar la comunicación en el proceso maestro. Para evitar este problema existen esquemas descentralizados donde la asignación de recursos es realizada por los distintos procesos, sin la existencia de un nodo central.

En este trabajo se proponen distintos enfoques para resolver el problema del balanceo de carga en aplicaciones distribuidas organizadas bajo una filosofía Master-Worker. Cada enfoque ha sido implementado y testeado para una amplia variedad de casos de estudio. El objetivo principal de esta investigación es evaluar los desempeños de las distintas estrategias a fin de establecer sus alcances, limitaciones y rango de aplicación.

2. ARQUITECTURAS

En esta sección se describen los enfoques de balanceo de carga propuestos en este trabajo. En primer término, se diseñó una arquitectura básica orientada a objetos a partir de la cual se fueron

implementando los distintos esquemas. El pasaje de mensajes se realizó a través de una clase abstracta encargada de la comunicación y sincronización. Los métodos de esta clase están vinculados dinámicamente a la librería PVM (Sunderam, 1990). No obstante, la extensibilidad y flexibilidad de esta implementación permite enlazar en forma dinámica esta clase con otras librerías de pasaje de mensajes (por ejemplo MPI).

2.1 Arquitectura Secuencial

Antes de describir la arquitectura distribuida básica, resulta necesario presentar el esquema secuencial, cuya implementación servirá de base para las comparaciones y evaluaciones de speed-up de los distintos algoritmos propuestos.

El funcionamiento de la arquitectura secuencial consiste de un ciclo. El *Manager* crea una tarea y la envía al *Worker*, quien luego de ejecutarla, devuelve su resultado al *Manager*. Este ciclo se repite hasta que no haya mas tareas para ejecutar. La *Figura 1* muestra el esquema básico de esta arquitectura, donde los rectángulos representan entidades activas, mientras que las elipses están asociadas a entidades pasivas. Estas convenciones se conservarán a lo largo de este trabajo.

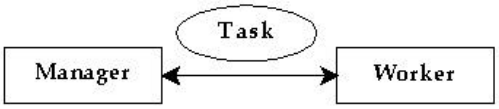

Figura 1

Como se observa en este gráfico solo existe un *Manager* y un *Worker*, y la ejecución de los mismos se realiza sobre una misma máquina.

2.2 Arquitectura Distribuida Básica

En términos generales, una arquitectura distribuida puede dividirse en dos partes: el *Manager* y el *Worker*, el primero se encarga de generar tareas y enviarlas a los *Workers*, los cuales las ejecutan, retornando los resultados al *Manager*. Los siguientes algoritmos describen el funcionamiento del *Manager* y el *Worker* respectivamente:

Manager:
1- Mientras haya tareas repetir 2 a 4:
  2- Generar una tarea
  3- Asignarle la tarea a un Worker
  4- Obtener el resultado

Worker:
  1- Obtener una tarea
  2- Ejecutar la tarea
  3- Devolver el resultado
4- Repetir 1 a 3 hasta que no existan tareas

Este esquema se muestra en la *figura 2*, donde la nube representa el medio de comunicación.

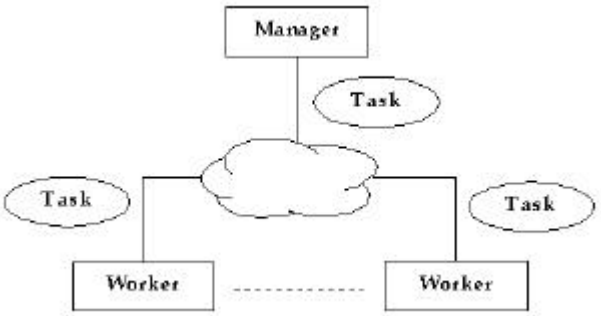

Figura 2

Para la arquitectura distribuida básica propuesta en este trabajo, la cual se muestra en la *figura 3*, el paso 3 del algoritmo del *Manager* es realizado por un *TaskManager*, el cual se encarga de administrar las tareas del *Manager*; mientras que el paso 1 del algoritmo del *Worker* es realizado por un *TaskCache*, el cual obtiene y almacena las tareas que deben ser ejecutadas por el *Worker*.

En este punto, es necesario aclarar que de ahora en adelante cuando mencionemos al *Manager*, podemos referirnos al objeto de la clase *Manager* o al conjunto de objetos que conforman y asisten al *Manager* en el balanceo de carga, la diferencia dependerá del contexto donde se utilice. Lo mismo ocurrirá para el *Worker*.

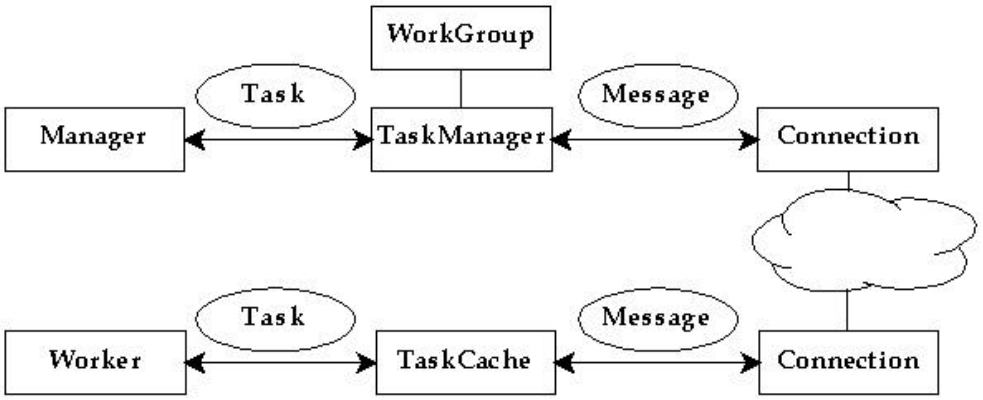

Figura 3

El *Manager* solo realiza dos acciones: generar tareas (*Tasks*) para entregárselas al *TaskManager*, y pedir al *TaskManager* el resultado de la ejecución de una tarea. Luego el *TaskManager* se encargará de distribuir las tareas entre los *Workers* de acuerdo a alguna política, para realizar dicha labor el mismo trabajara en conjunto con el *TaskCache* intercambiando mensajes por medio de un objeto de clase *Connection*. Esta clase es encargada de la comunicación y sincronización de mensajes.

El *Worker* pide una tarea por vez al *TaskCache*, la ejecuta y devuelve el resultado de la ejecución al mismo. Luego, el *TaskCache* se encarga que el resultado llegue al *TaskManager*.

Por otra parte, el *WorkGroup* es el que mantiene la información de todos los *Workers* en la arquitectura, y puede ser usado por el *TaskManager* para realizar la distribución de tareas.

Como se observa, la forma en que se produce el intercambio de tareas y resultados es transparente para el *Manager* y los *Workers*, ya que dichas acciones son realizadas por el *TaskManager* y los *TaskCache*. Luego, estas dos clases son las que implementan las políticas de balanceo de carga.

2.3 Arquitectura Estática

A continuación se describe la arquitectura estática, la cual distribuye todas las tareas en forma equitativa entre todos los Workers antes de comenzar el cómputo. El término equitativo se refiere a que cada *Worker* recibe la misma cantidad de tareas, o sea, si tenemos w workers y t tareas, la cantidad de tareas que recibirá cada worker será t/w.

La *figura 4* muestra el modelo estático, el cual presenta algunos cambios respecto de la arquitectura básica. En primer lugar, como las clases *Connection* y *WorkGroup* de la *figura 3* son abstractas, para la implementación de los distintas políticas de balanceo de carga se definieron las subclases *PVMConnection* y *PVMWorkGroup* utilizando la librería PVM.

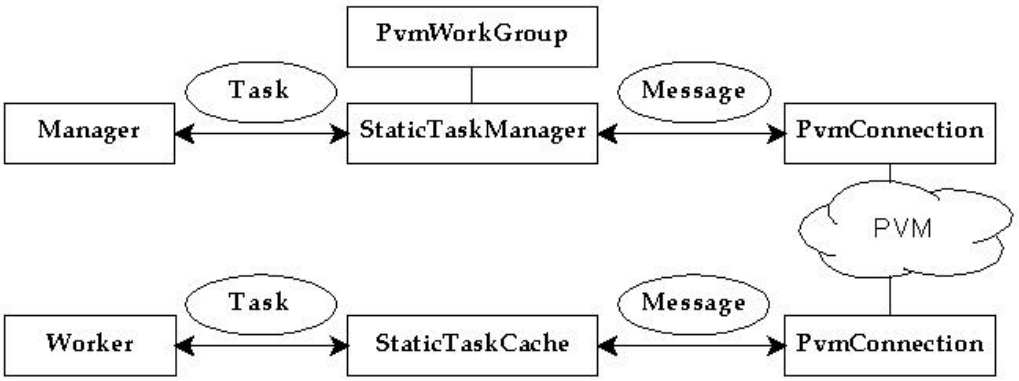

Figura 4

El funcionamiento del esquema se divide en dos partes. Por una lado, el *Manager* genera una tarea y se la pasa al *StaticTaskManager*, este obtiene del *WorkGroup* el próximo *Worker* de la secuencia de *Workers* definida estáticamente, y mediante *Connection* le envía la tarea al *StaticTaskCache* del *Worker* seleccionado. Esto se repite hasta que el *Manager* no genere nuevas tareas, etapa en la cual le pide al *StaticTaskManager* los resultados de la ejecución de todas las tareas creadas. Al realizar esto, el *StaticTaskManager* le envía a todos los *Workers* un mensaje indicándoles que ya no hay más tareas, para luego quedar en un estado de escucha esperando que los Workers le envíen los resultados. Una vez recibidas todas las respuestas, se las pasa al *Manager* en una lista, y finaliza. Por otra parte, cada *Worker* le pide una tarea al *TaskCache* y si este tiene alguna se la asigna. En caso contrario, el *Worker* se queda esperando hasta que exista alguna. Cada tarea enviada por el *TaskCache* es ejecutada por el *Worker* y el resultado es pasado al *TaskCache*, el cual se encarga de enviarlo al *TaskManager*.

El *TaskCache* al principio se queda esperando por tareas, las cuales se va almacenando en una cola para luego pasárselas al *Worker*. Asimismo, las respuestas retornadas por su *Worker* son encoladas en otra estructura. Cuando el *TaskCache* recibe un mensaje del *TaskManager* indicando que no hay más tareas, cambia de estado y empieza a enviar los resultados de las ejecuciones al *TaskManager* hasta enviar el último, momento en el cual termina.

Finalmente, cabe aclarar que se considera que el *Manager* y el *TaskManager* trabajan en forma independiente y concurrente, para lograrlo, los mismos fueron implementados utilizando hilos y semáforos como herramientas de concurrencia y sincronización respectivamente. Lo mismo acontece con las clases *Worker* y *TaskCache*. Otro detalle importante es que todas las esperas producidas por la sincronización no consumen ciclos de CPU, por lo que no afectan el desempeño de los procesos relacionados. Estos aspectos de implementación fueron llevados a cabo en todos los esquemas de balanceo de carga aquí propuestos.

2.4 Arquitectura Dinámica

En la arquitectura dinámica, a diferencia del caso anterior, no se saben cuantas tareas se van a ejecutar en cada *Worker*, esto depende de la capacidad de cómputo de los Workers y las características de cada tarea.

En la figura 5 se muestra el esquema dinámico. El aspecto principal de este enfoque es que el *Worker* demanda tareas al *Manager* cada vez que esta desocupado.

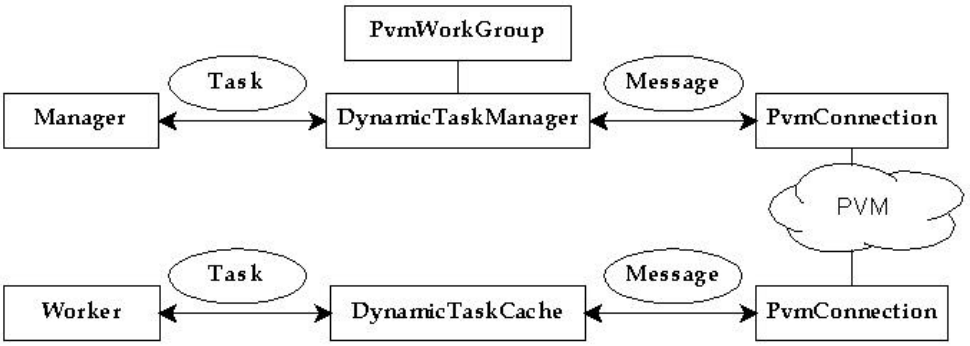

Figura 5

Como en el caso anterior el *Manager* genera una tarea, se la pasa al *TaskManager* y este la almacena en una cola. Luego, cuando el *Manager* termina de crear las tareas, le pide el resultado al *TaskManager* y se queda esperando.

El *TaskManager* cumple un rol pasivo respecto de su comunicación con los *TaskCache*. Cada vez que le solicitan una tarea, el *TaskManager* toma una de su cola y se la envía al *TaskCache* del *Worker* que la pidió, si no tiene más tareas, le envía un mensaje indicando que no hay más. Si recibe un mensaje que contiene un resultado lo agrega a una lista para luego pasársela al *Manager*.

Por otro lado, cada vez que un *Worker* está ocioso, le pide una tarea al *TaskCache*, este le envía un mensaje al *TaskManager* pidiéndole una tarea, este le responde, y si existe una tarea se la pasa al *Worker*, en caso contrario termina. Una vez ejecutada una tarea, el *Worker* le pasa el resultado al *TaskCache* y le pide otra, con lo cual el *TaskCache* envía el resultado al *TaskManager* y le pide otra tarea.

Nótese que esta arquitectura tiene la desventaja de que el *Worker* y el *TaskCache* tienen un instante de tiempo ocioso que se produce cuando el *TaskCache* envía y recibe mensajes. Esto motivó el diseño de una variante este enfoque, el cual se presenta en la siguiente sección.

2.5 Arquitectura Dinámica con Cola

La arquitectura dinámica con cola trata de resolver el problema planteado anteriormente, es decir, trata de que nunca se produzca un estado en el cual el *Worker* y el *TaskCache* estén ociosos. Para lograr esto, el *TaskCache* trae tareas por anticipado y las mantiene en una cola, con lo cual el *Worker* nunca tiene que esperar a que llegue una tarea del *Manager*.

En la figura 6, se presenta el esquema dinámico con cola.

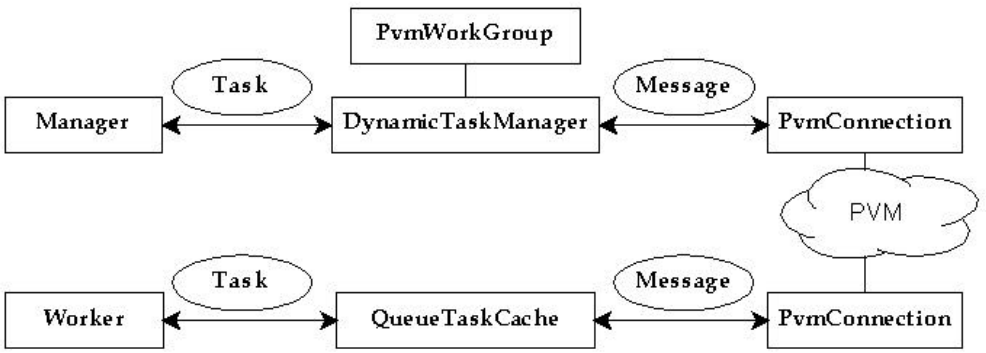

Figura 6

Antes de explicar el funcionamiento es necesario hacer notar que en este modelo el *TaskManager* es el mismo del modelo dinámico, esto se debe a que la variación propuesta en esta arquitectura solo afecta al *TaskCache*. El *Worker* tampoco sufre cambios de implementación. Al igual que antes, pide tareas, las ejecuta y devuelve sus resultado. La única diferencia está en que los tiempos de espera se reducen drásticamente.

Para explicar el funcionamiento del *QueueTaskCache* es necesario definir dos constantes que llamaremos C_{max} y C_{min}; donde la primera describe la máxima cantidad de tareas que puede almacenar la cola, y la segunda indica una cota inferior a partir de la cual el *QueueTaskCache* debe empezar a pedir tareas nuevamente.

Cuando *QueueTaskCache* empieza a trabajar, comienza a pedir tareas al *DynamicTaskManager* hasta que la cola contenga C_{max} tareas. En este punto, deja de pedir tareas y se queda en un estado de espera. El *Worker* puede solicitar tareas en cualquier momento, ejecutarlas y luego retornar el resultado al *QueueTaskCache*. Es importante destacar, que como el *QueueTaskCache* va trayendo tareas por anticipado, generalmente la cola de tareas no está vacía, y por lo tanto, los tiempos de espera del *Worker* se reducen significativamente.

El *Worker* pide y ejecuta, hasta un momento en el cual el tamaño de la cola es C_{min}, en ese instante, el *QueueTaskCache* se reactiva y vuelve a pedir tareas al *DynamicTaskManager* hasta alcanzar nuevamente el tamaño C_{max}. Esto se repite hasta que recibe el mensaje de que no hay mas tareas, momento en el cual se queda esperando a que la cola se vacíe, para luego terminar.

3. ANÁLISIS COMPARATIVO DE DESEMPEÑO

El primer paso del análisis consistió en plantear las características de las tareas a emplear como casos de estudio. Como los aspectos más relevantes para la comparación de desempeño entre las distintos esquemas son los tiempos de cómputo de cada tarea y los tiempos consumidos en la transmisión de las mismas, se caracterizó cada tarea con una complejidad, la cual afecta al tiempo de ejecución; y un tamaño de dato, el cual afecta el tiempo de transmisión y a su vez el tiempo de ejecución. También se asoció a cada tarea una constante que multiplica a la complejidad. Luego, las variables asociadas a cada tarea son:

- **Orden(o)** Define el tipo de algoritmo que se va a aplicar sobre el dato, cada algoritmo esta definido de acuerdo a su orden considerándose los siguientes casos:
    - Logarítmicas.
    - Lineales.
    - Cuadráticas.
    - Cuadráticas Logarítmicas.
    - Cúbicas.
- **Data Long(dl)** Define la longitud del dato sobre el cual se va a aplicar una tarea.
- **Constante(c)** Define una constante que multiplica al orden.

El siguiente aspecto que se consideró fue el *Manager*, en este caso para que la simulación fuera más cercana a la realidad se asoció al mismo un retardo entre la creación de las distintas tareas, de esta forma el mismo puede provocar retardos en los *Workers* si estos piden tareas y el *Manager* todavía no las creo. Otros aspectos que se consideraron son: la cantidad de *Workers* (que generalmente son uno por nodo) y la cantidad de tareas que genera el *Manager* . Además, también se tuvo en cuenta como se definirían los datos asociados a cada tarea, para que los mismos no fueran homogéneos. Para cada variable se definió una varianza y para el orden se distinguió entre

dos casos: homogenas (todas las tareas del mismo orden) y heterogeneo (tareas de distintos órdenes). De este modo, el manager cuenta con los siguientes parámetros:

- **Workers(cw)** Define la cantidad de workers que se van a usar en la simulación.
- **Tasks(ct)** Define la cantidad de tareas que se van a usar.
- **Orden(mo)** Define el orden de las tareas, pudiendo ser uno de los ya definidos en las tareas (homogéneo) o generados al azar (heterogéneo).
- **Delay(d)** Define el retardo entre la creación de cada tarea.
- **Var Delay(vd)** Define una varianza para el retardo.
- **Data Long(mdl)** Define la longitud del dato promedio, sobre la cual se va a aplicar la varianza, para crear las tareas.
- **Constante(mc)** Define la constante promedio, sobre la cual se va a aplicar la varianza, para crear las tareas.
- **Var Data Long(mvdl)** Define varianza de la longitud del dato.
- **Var Constante(mvc)** Define una varianza para la constante.

3.1 Evaluación de desempeño

La simulación se realizó usando una red de 10Mbits conformada por ocho procesadores Pentium de 200Mhz. Para obtener cada resultado se realizó un mínimo de tres corridas, calculándose luego el promedio de las mismas. En cada caso de estudio se consideraron las siguientes alternativas:

Estática: muestra los valores obtenidos para la implementación de la arquitectura estática.

Dinámica: muestra los valores obtenidos para la implementación de la arquitectura dinámica.

Cola 1: muestra los valores obtenidos para la implementación de la arquitectura dinámica con cola, tomando como valores C_{min}=1 y C_{max}=1.

Cola 2: muestra los valores obtenidos para la implementación de la arquitectura dinámica con cola, tomando como valores C_{min}=1 y C_{max}= 2.

Cola 3: muestra los valores obtenidos para la implementación de la arquitectura dinámica con cola, tomando como valores C_{min}=2 y C_{max}= 4.

Para calcular el tiempo secuencial se uso una implementación de la arquitectura secuencial, la cual solamente crea y ejecuta tareas, eliminando de esta forma los tiempos producidos por el pasaje de mensajes entre procesos.

Luego parar cada alternativa se muestra el tiempo de ejecución (expresados en segundos), el speed-up y la eficiencia obtenidos, los cuales se muestran en columnas contiguas en cada esquema, respectivamente.

Caso de estudio 1

Se usaron los siguientes valores:

ct	mo	d	vd	mdl	mvdl	mc	mvc
500	*Lineales*	*1000*	*10*	*1000*	*100*	*1000*	*10*

Cant.Proc.	Estática			Dinámica			Cola 1			Cola 2		
8	7.7	4.62	57.71 %	5.98	5.94	74.31 %	5.74	6.19	77.42 %	5.64	6.30	78.79 %
4	12.37	2.87	71.85 %	10.55	3.37	84.24 %	10.32	3.44	86.12 %	10.46	3.40	84.97 %
2	21.58	1.65	82.37 %	21.05	1.69	84.44 %	20.27	1.75	87.69 %	20.24	1.76	87.82 %
Secuencial	35.55											
Efic. Prom.	*70.64%*			*81%*			*83.74%*			*83.86%*		

Como se observa en la tabla, al trabajar con tareas de corta duración, el modelo dinámico con cola obtiene mejores resultados que los demás. Si consideramos la alternativa *Cola 2*, vemos que para tamaños de cola más grande los tiempos mejoran. Además, cabe destacar que los tiempos obtenidos son mejores que los logrados con el modelo estático, a pesar de tratarse de tareas y procesadores homogéneos en donde los esquemas estáticos usualmente resultan superiores.

Caso de estudio 2

Se usaron los siguientes valores:

ct	mo	d	vd	mdl	mvdl	mc	mvc
100	*Cuadráticas*	*1000*	*10*	*100*	*10*	*1000*	*10*

Cant.Proc.	Estática			Dinámica			Cola 1			Cola 2		
8	7.01	7.46	93.28 %	7.05	7.42	92.75 %	7.04	7.43	92.88 %	7.15	7.32	91.45 %
4	13.62	3.84	96.02 %	13.56	3.86	96.44 %	13.59	3.85	96.23 %	13.62	3.84	96.02 %
2	25.56	2.05	102.33 %	26.75	1.96	97.78 %	26.7	1.96	97.96 %	26.82	1.95	97.52 %
Secuencial	52.31											
Efic. Prom.	*97.21%*			*95.66%*			*95.69%*			*95%*		

En este caso el modelo estático obtiene mejores resultados, lo cual era predecible. Si comparamos el modelo dinámico simple con los esquemas con colas, se aprecia que con colas más chica obtiene una mejor eficiencia promedio, mientras que con colas más grandes se produce un mayor desbalance entre los Workers.

Caso de estudio 3

Se usaron los siguientes valores:

ct	Mo	D	vd	mdl	mvdl	mc	mvc
100	*Cuadráticas Logarítmicas*	*1000*	*10*	*100*	*10*	*1000*	*10*

Cant.Proc.	Estática			Dinámica			Cola 1			Cola 2		
8	48.65	7.5	93.76 %	47.98	7.61	95.07 %	47	7.76	97.05 %	48.6	7.51	93.86 %
4	92.60	3.94	98.52 %	93.03	3.92	98.06 %	93.7	3.89	97.36 %	94.3	3.87	96.74 %
2	184.8	1.97	98.73 %	184.4	1.98	98.95 %	184.63	1.98	98.82 %	186.34	1.96	97.92 %
Secuencial	364.91											
Efic. Prom.	*97%*			*97.36%*			*97.74%*			*96.17%*		

Al observar los resultados, se aprecia que el modelo dinámico obtuvo mejores tiempos, mientras que el modelo con cola 1 obtiene la mejor eficiencia, esto se debe a que este último produce un mejor escalamiento. Nuevamente se vuelve a dar la situación que para colas grandes el modelo empeora.

Caso de estudio 4

Se usaron los siguientes valores:

ct	mo	d	vd	mdl	mvdl	mc	Mvc
50	*Azar*	*1000*	*10*	*100*	*10*	*1000*	*10*

Cant.Proc.	Estática			Dinámica			Cola 1			Cola 2		
8	256.59	3.15	39.39 %	136.87	5.91	73.85 %	134.64	6.01	75.07 %	231.84	3.49	43.60 %
4	354.96	2.28	56.95 %	242.09	3.34	83.50 %	251.23	3.22	80.46 %	262.03	3.09	77.15 %
2	556.49	1.45	72.65 %	422.57	1.91	95.68 %	413.87	1.95	97.69 %	417.77	1.94	96.78 %
Secuencial	808.6											
Efic. Prom.	*56.33%*			*84.34%*			*84.41%*			*72.51%*		

Este caso es el más interesante de todos ya que trabajamos con tareas heterogéneas. Como era de esperarse el modelo estático sufre una fuerte pérdida de desempeño respecto de los restantes enfoques. Si comparamos los otros dos modelos, nuevamente el modelo con cola más chica logra la mejor eficiencia promedio.

Caso de estudio 5

Se usaron los siguientes valores:

ct	mo	d	vd	mdl	mvdl	mc	mvc
500	*Cuadráticas*	*1000*	*10*	*100*	*10*	*1000*	*10*

Cant.Proc.	Estática			Dinámica			Cola1			Cola 3		
8	33.81	7.72	96.54 %	33.48	7.80	97.51 %	33.52	7.79	97.39 %	33.57	7.78	97.25 %
4	66.26	3.64	98.53 %	66.51	3.93	98.16 %	66.37	3.93	98.37 %	66.42	3.93	98.29 %
2	132.25	1.97	98.73 %	132.47	1.97	98.57 %	132.08	1.98	98.86 %	132.12	1.98	98.82 %
Secuencial	261.14											
Efic. Prom.	*97.93%*			*98.08%*			*98.21%*			*98.12%*		

Acá se aprecia que cuando el número de tareas aumenta, el desempeño del modelo estático decae, mientras que mejoran los resultados obtenidos con los modelo dinámico con cola.

Caso de estudio 6

Se usaron los siguientes valores:

ct	mo	d	vd	mdl	mvdl	mc	mvc
100	*Cuadráticas*	*1000*	*10*	*200*	*10*	*1000*	*10*

Cant.Proc.	Estática			Dinámica			Cola 1			Cola 3		
8	28.64	7.57	94.59 %	28.53	7.60	94.96 %	28.44	7.62	95.25 %	28.82	7.52	94.00 %
4	55.08	3.93	98.37 %	55.40	3.91	97.80 %	55.48	3.91	97.66 %	55.31	3.92	97.95 %
2	109.63	1.98	98.84 %	109.81	1.97	98.68 %	109.84	1.97	98.65 %	109.81	1.97	98.68 %
Secuencial	216.72											
Efic. Prom.	*97.26%*			*97.15%*			*97.19%*			*96.88%*		

Aquí el modelo estático obtiene los mejores tiempos, y se aprecia que el esquema con cola C_{min}=2 y C_{max}= 4 sufre de un fuerte desbalanceo de carga.

Caso de estudio 7

Se usaron los siguientes valores:

Ct	mo	d	vd	mdl	mvdl	mc	mvc
500	*Azar*	*1000*	*10*	*100*	*10*	*1000*	*10*

siendo las tareas generadas aleatoriamente de tipo Logarítmicas, Lineales y Cuadráticas.

Cant.Proc.	Estática			Dinámica			Cola 1			Cola 3		
8	14.28	6.40	80.06 %	12.41	7.37	92.13 %	12.41	7.37	92.07 %	12.63	7.24	90.52 %
4	27.84	3.28	82.11 %	24.12	3.79	94.79 %	23.91	3.82	95.62 %	24.09	3.80	94.89 %
2	47.80	1.91	95.65 %	47.57	1.92	96.11 %	47.16	1.94	96.95 %	47.15	1.94	96.97 %
Secuencial	91.44											
Efic. Prom.	*85.94%*			*94.34%*			*94.88%*			*94.13%*		

Al repetir las pruebas con tareas heterogéneas, pero de complejidad menor que las del caso de estudio 4, notamos que el modelo dinámico con cola 1 aumenta la diferencia respecto del modelo dinámico simple.

Caso de estudio 8

Se usaron los siguientes valores:

Ct	mo	d	vd	mdl	mvdl	mc	mvc
200	*Azar*	*1000*	*10*	*300*	*10*	*1000*	*10*

siendo las tareas generadas aleatoriamente de tipo Logarítmicas, Lineales y Cuadráticas.

Cant.Proc.	Estática			Dinámica			Cola 1		
8	60.16	5.99	74.87 %	46.82	7.70	96.21 %	50.69	7.11	88.86 %
4	108.05	3.34	83.38 %	92.29	3.90	97.61 %	96.29	3.74	93.56 %
2	195.39	1.84	92.21 %	182.86	1.97	98.53 %	182.79	1.97	98.57 %
Secuencial	360.35								
Efic. Prom.	*83.49%*			*97.45%*			*93.66%*		

Acá se observa que en problemas heterogéneos, al incrementar el tamaño de las tareas, el modelo dinámico supera a los esquemas con cola ampliamente.

4. CONCLUSIONES

Las conclusiones se pueden organizar para dos grandes grupos: tareas homogéneas y heterogéneas. Asimismo, cada caso puede subdividirse en función de la cantidad de tareas y la duración de las mismas.

Para el caso homogéneo, cuando se trabaja con pocas tareas de orden cuadrático, el modelo estático logra el mejor desempeño. En las restantes situaciones, el modelo con cola obtiene una mejor eficiencia que los demás para un tamaño de cola con C_{min}=1 y C_{max}=1. Cabe destacar que para tareas de orden lineal con C_{min}=1 y C_{max}=2 se alcanzaron los mejores resultados.

Al considerar tareas heterogéneas, el modelo estático no es una alternativa viable, tal como podía esperarse. En este caso las mejores opciones son el modelo dinámico y el dinámico con cola parametrizada en C_{min}=1 y C_{max}=1. En general, se observó que este último obtiene un mejor desempeño para tareas de corta duración, mientras que el dinámico logra mejores tiempos cuando se trabaja con tareas más grandes.

En general, independientemente del tipo de problema, se observó que el esquema de balanceo de carga dinámico con cola C_{min}=1 y C_{max}=1 siempre logra uno de los dos mejores desempeños. La razón radica en que este modelo evita que el Worker se encuentre en un estado de ocio debido al almacenamiento anticipado de una tarea. Por otra parte, si se trabaja con colas más grandes ($C_{max} > C_{min} \geq 1$), se puede producir un desbalance de carga, lo cual tiende a causar un efecto similar al del modelo estático. De todo esto se desprende nuestra recomendación de emplear un modelo dinámico de cola con C_{min}=1 y C_{max}=1 cuando no se dispone de un conocimiento preciso de la naturaleza de las tareas, dado que en promedio este modelo constituye el mejor esquema de balanceo de carga.

5. REFERENCIAS

Buyya R., "High Performance Cluster Computing", Volume 2, Prentice Hall, **1999**.

Chalmer A. y Tidmus J., "Practical Parallel Processing", Thomson Computer Press, **1996**.

Chaudhuri P., "Parallel Algorithms: Design and Analysis", Prentice Hall, **1992**.

Sunderam V., "PVM: A Framework for Parallel Distributed Computing", Concurrency: Practice & Experience, 2, 4, 315-339, **1990**.

Tinetti F.G. y De Giusti A., "Procesamiento Paralelo. Conceptos de Arquitecturas y Algoritmos", Editorial Exacta, **1998**.

Nombre de archivo: CACIC 2001 - Comparación de Esquemas de Balanceo de Carga (versión final).doc
Directorio: C:\WINDOWS\DESKTOP
Plantilla: C:\WINDOWS\Profiles\ponzoni\Application Data\Microsoft\Plantillas\Normal.dot
Título: Load Balancing
Asunto:
Autor: Diego Oscar Scarpa
Palabras clave:
Comentarios:
Fecha de creación: 8/15/01 4:34
Cambio número: 8
Guardado el: 8/15/01 4:55
Guardado por: NN
Tiempo de edición: 22 minutos
Impreso el: 8/15/01 4:55
Última impresión completa
Número de páginas: 12
Número de palabras: 3,988 (aprox.)
Número de caracteres: 22,733 (aprox.)