Hash consistente como uma ferramenta para distribuição de tarefas em sistemas distribuídos reconfiguráveis

André Ribeiro da Silva Hélio Marcos Paz de Almeida Tiago Macambira,
Dorgival Olavo Guedes Wagner Meira Jr. Renato Antonio C. Ferreira
Laboratório e-Speed
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
{arsilva,helio,tmacam,dorgival,meira,renato}@dcc.ufmg.br

Resumo

Sistemas distribuídos que incluem processos replicados em várias máquinas têm que oferecer soluções para o problema de distribuição dinâmica de tarefas. No caso em que a aplicação requer a manutenção de algum de estado entre tarefas é preciso um esquema consistente de endereçamento. Isso é usualmente feito através de uma função de hash *tradicional, que entretanto não funciona bem em ambientes com reconfiguração dinâmica. Neste trabalho avaliamos o uso de* hash *consistente, uma forma de* hash *onde a função de distribuição se altera pouco com a alteração da sua faixa de operação, como uma solução para esse problema. Comparamos o desempenho de duas soluções de* hash *consistente com soluções tradicionais em termos da uniformidade do padrão de distribuição, do número de mensagens trocadas quando de uma reconfiguração e do padrão de comunicação envolvido. Nossos resultados mostram que* hash *consistente tem um overhead de comunicação muito menor que a solução tradicional (até 35 vezes em alguns testes), com um aumento aceitável da variância na distribuição de tarefas.*

1. Introdução

Sistemas distribuídos reconfiguráveis são caracterizados pela sua capacidade de variar o número de nós de processamento dinamicamente. Essa variação tem por objetivo adequar o sistema às demandas da aplicação, por exemplo, incluindo mais nós para atender à carga computacional crescente de uma aplicação, ou permitir que o mesmo se adeqüe a eventos de origem externa à aplicação, como por exemplo no caso da falha de nós. Tanto no caso da inclusão de novos nós quanto na sua adaptação a uma configuração com menos nós, caso se deseje que a aplicação continue seu processamento fazendo o melhor uso do sistema disponível, é preciso que ela seja capaz de se adaptar às mudanças.

Ambientes de processamento distribuído com tais características dinâmicas muitas vezes usam replicação de tarefas como forma de obter paralelismo: cada nó do sistema executa uma cópia do mesmo código sobre dados (tarefas) diferentes. Também conhecido como paralelismo de dados, esse modelo de programação se adapta de forma relativamente simples no caso de reconfigurações, simplesmente gerando novas cópias do código no caso de inclusões de nós ou trabalhando com o número reduzido resultante de uma falha.

Modelos de programação onde as tarefas executadas pelas réplicas do código são completamente independentes, como no caso de grupos de tarefas (*bag of tasks, BOT* [4]), por exemplo, se adaptam de forma bastante simples a reconfigurações: no caso de inclusões de nós, as novas réplicas podem simplesmente passar a receber tarefas do repositório de tarefas comum; no caso de falhas, as tarefas então sendo processadas por um nó que falhe podem simplesmente ser reiniciadas em outros nós do sistema [5]. Essa solução é utilizada, por exemplo, na implementação de sistemas distribuídos bem sucedidos como SETI@home [1] e Google [6], assim como na distribuição de tarefas em servidores WWW distribuídos [3].

Entretanto, outros tipos de aplicações que exploram paralelismo de dados exigem ambientes de execução onde a atribuição de tarefas a nós processadores permita a definição de afinidades: certos dados ou tarefas que compartilhem certas características devem ser sempre assinalados a uma mesma réplica para processamento. Tal restrição usualmente se relaciona a casos onde algum tipo de estado precisa ser preservado entre tarefas, ou ao longo do processamento de diversas unidades de dados que fluem pelo sistema. Um exemplo de tais aplicações são os *clusters* de servidores de comércio eletrônico, onde requisições de um cliente devem ser sempre direcionadas para o mesmo servidor,

a fim de simplificar a manutenção do histórico da sessão (por exemplo, o conteúdo do "carrinho de compras") [17]. Outro exemplo são aplicações distribuídas iterativas, onde dados são produzidos e consumidos dinamicamente pelas réplicas que executam um *loop* de processamento, muitas vezes devendo atender a restrições como dependências de dados ou atualizações de estado dependente dos dados. Um ambiente que implementa esse modelo de programação é, por exemplo, o Formigueiro (*Anthill*), projeto no qual se insere este trabalho [7].

Em tais sistemas é preciso que haja técnicas que garantam uma forma eficiente de se assinalar dados aos nós responsáveis pelo seu processamento. Uma solução comumente utilizada é a definição de uma função de *hash* tradicional aplicada a uma parte dos dados que possa ser usada para esse fim. Dado um conjunto fixo de n nós, todos os dados que se refiram a um dado estado i identificado por um padrão ou *label* existente nos dados será sempre mapeado para o mesmo nó. Uma mudança no número de réplicas de processamento leva a uma mudança da função de *hash* .

Entretanto, em aplicações com restrições de afinidades de dados ou manutenção de estado ocorre um problema quando o sistema distribuído é sujeito a reconfigurações durante a execução. Se novos nós são acrescentados para atender a um aumento da carga de processamento, por exemplo, o reassinalamento de tarefas para aqueles nós vai exigir que o estado das tarefas a eles assinalados lhes seja transferido antes que o processamento continue. Da mesma forma, se um nó é retirado do sistema, pelo menos as tarefas que lhe eram atribuídas terão que ser assinaladas a outros nós e o estado já acumulado para as mesmas precisará ser distribuído para outros nós.

Nesse contexto, soluções de *hash* tradicionais constituem um problema, pois uma mudança no número de nós do sistema pode levar a um reassinalamento de quase todas as tarefas já em execução no sistema. Isso causa um volume de tráfego excessivamente alto entre os nós do sistema, quando praticamente todas as tarefas devem ter seu estado transferido para novos nós.

Neste trabalho avaliamos esse impacto para duas soluções de assinalamento por *hash* habituais e propomos o uso de soluções baseadas no conceito de *hash consistente* [9] para amenizar o problema observado. Os resultados indicam que as soluções propostas podem reduzir drasticamente o custo de reassinalamento de tarefas. Para isso, o restante deste trabalho está organizado da seguinte forma: a seção a seguir introduz os principais trabalhos relacionados; em seguida, a seção 3 discute a aplicação de soluções de *hash* tradicional ao problema de assinalamento de tarefas e compara essas técnicas ao uso de *hash consistente*. As seções 4 e 5 apresentam a metodologia utilizada e os resultados, respectivamente, onde os ganhos da solução proposta são claramente observados. Finalmente, a seção 6 apresenta algumas conclusões e propostas de trabalhos futuros.

2. Trabalhos relacionados

O problema de reconfiguração de sistemas com manutenção de estado foi bastante estudado no contexto de *clusters* de servidores WWW, especialmente com relação a sistemas de comércio eletrônico [17, 3]. Nesses casos, soluções baseadas em *hash* tradicional são complementadas com uma camada de persistência em banco de dados que faz a manutenção e transferência de estado quando necessário. Dadas as características do sistema, reconfigurações em geral não causam problemas maiores: novos servidores recebem novas requisições, falhas de servidores levam ao término das sessões a eles relacionadas.

Hash consistente foi proposto originalmente por Karger *et alli* [9]. Uma análise detalhada das propriedades teóricas das famílias de funções de *hash* consistente podem ser encontrada naquele trabalho. O enfoque original focava sistemas de caches WWW, visando permitir um assinalamento de URLs a caches que reduzisse o impacto de mudanças no conjunto de caches ativas. Nesse contexto, uma característica importante estudada foi a capacidade de sistemas baseados em *hash* consistente de operar sob condições onde os participantes possuem apenas visões parciais do conjunto de nós ativos [10]. No caso tratado neste artigo essa característica não é relevante, pois o processamento de uma aplicação exige um assinalamento determinístico e inequívoco de tarefas a nós de processamento.

A capacidade do modelo de *hash* consistente se adaptar a visões variáveis do conjunto de participantes ao mesmo tempo que garante um mapeamento confiável entre chaves e nós destino torna-o uma solução interessante para redes *peer-to-peer* [2]. Nesses sistemas, o processo de assinalamento da chave ao nó apropriado toma a forma de uma consulta recursiva ao longo da rede [12], dando origem ao termo "tabela de *hash* distribuída" (*Distributed Hash Tables, DHT*) [14]. Exemplos de sistemas que utilizam esse tipo de recurso são as redes CAN [13], CHORD [16] e PASTRY [15].

A maior parte dos trabalhos até o momento sobre *hash* consistente e DHTs consideram um universo onde todas as chaves existentes são equivalentes em termos de sua demanda por recursos, por exemplo, a capacidade de processamento associada a cada tarefa. Novas soluções começam a ser consideradas para casos onde isso não é verdade [8]. Em aplicações dependentes de dados, por exemplo, uma solução que garanta balanceamento de carga deve levar isso em conta.

3. Técnicas de distribuição utilizando *hash*

Antes de avaliarmos o comportamento das técnicas de reassinalamento de tarefas é importante entendermos os princípios das mesmas.

3.1. Hash tradicional

Como discutido anteriormente, uma solução comumente utilizada para a distribuição de tarefas em *clusters* seguindo um critério determinístico é a aplicação de uma função de *hash* tradicional a uma chave identificadora de cada tarefa que mapeie a chave ao intervalo $[0..n-1]$ utilizado para identificar os n nós disponíveis em um determinado momento. A função mais tradicional é da forma $h(x) = x \mod n$, onde x seria um mapeamento da chave escolhida para o domínio dos números inteiros com boa distribuição dentro do espectro de valores possíveis.

Essa função tem como qualidades a simplicidade de sua implementação e uma distribuição das chaves bastante uniforme entre os nós do sistema. Entretanto, para fins de implementação de um sistema reconfigurável, essa função apresenta um grave problema, ilustrado pela Fig. 1 para o caso da inserção de um novo nó no sistema (elevando o número de nós para $n+1$).

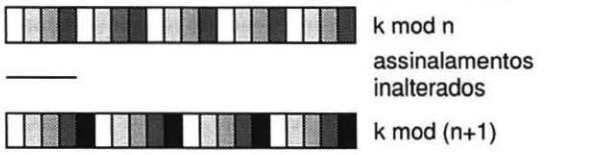

Figura 1. Redistribuição usando *hash* com resto da divisão

Na figura vê-se um espaço de 20 chaves, mapeado inicialmente para quatro nós (identificados por diferentes tons de cinza), que é redistribuído com a inclusão de mais um nó. É fácil ver que, para esse conjunto, apenas quatro chaves permanecem nos nós originais após a reconfiguração. Todas as demais são assinaladas a novos destinos. Em um sistema distribuído, isso levaria a um enorme volume de mensagens desnecessárias, pois se considerarmos a situação ideal para manter a distribuição uniforme, bastaria a cada nó enviar apenas uma chave para o novo nó para manter o equilíbrio.

Idealmente, para n nós e k chaves, a inclusão de um novo nó levaria à migração de $k/(n+1)$ chaves para o novo nó. Entretanto, pode-se mostrar que para uma função de *hash* tradicional do tipo $k \mod n$, das k/n chaves em cada nó antes da reconfiguração, apenas $k/(n(n+1))$ permanecerão no mesmo nó, ou seja, $k/(n+1)$ chaves de cada nó existente antes da reconfiguração serão reassinaladas. Logo, $nk/(n+1)$ chaves serão reassinaladas no total, gerando um tráfego n vezes maior que o mínimo necessário[1]. Em média, cada nó envia uma fração de $1/n$ das suas chaves para cada um dos outros nós, em um padrão de comunicação bastante uniforme de todos para todos. Esse tráfego é bastante significativo, ainda mais à medida que o sistema cresce (se n aumenta cada vez uma fração maior das chaves muda de destino, tendendo a 100 % quando n tende a infinito). Esse é o problema que se pretende resolver com o uso de *hash* consistente.

Antes de discutirmos a utilização de *hash* consistente, é interessante verificar também outra função de *hash* tradicional que tem sua utilização nesse tipo de problemas: $h(x) = x \texttt{ div } n$, cujo comportamento é ilustrado na Fig. 2. Essa função apresenta características semelhantes à anterior, porém com uma distribuição que pode ser sensivelmente mais irregular caso o mapeamento das chaves para inteiros não seja muito uniforme. Se esse não for o caso, esse mapeamento apresenta como característica interessante o fato de que chaves com um prefixo comum tendem a ser assinaladas a um mesmo nó, ou a nós consecutivos. Em certos padrões de comunicação em sistemas distribuídos, essa é uma característica desejável para reduzir custos de comunicação[11].

Figura 2. Redistribuição usando *hash* com divisão inteira

Como pode ser visto pela figura, esse assinalamento também tem como elemento interessante um maior número de chaves que permanecem em seus nós originais após a reconfiguração. Entretanto, uma análise mais cuidadosa indica que cada nó i (i variando de 0 a n-1) deve transferir para o nó seguinte na configuração resultante um total de $(i+1)k/(n(n+1))$. Pela figura isso pode ser visto no padrão onde o último nó antes da reconfiguração mantém apenas uma pequena fração das chaves inicialmente a ele atribuídas, enquanto cada nó anterior retém uma fração maior das suas chaves após a inclusão. Ao contrário do tráfego com a função de *hash* anterior, o tráfego nesse caso tem um padrão mais regular, onde cada nó transfere chaves apenas para o seu vizinho imediato. Entretanto, esse tráfego

1 Uma análise semelhante vale para o caso da redução do valor de n

chega a ser tão intenso quanto a soma das movimentações de um nó no esquema anterior (considerando-se o último nó da cadeia). A regularidade da comunicação pode ser um fator positivo em certas arquiteturas, mas a redução de tráfego total não é tão significativa em relação à anterior: a cada novo nó incluído, metade das chaves são transferidas entre os nós.

3.2. Hash consistente

Como mencionado anteriormente, Hash consistente foi proposto como uma solução para o problema de reassinalamento de chaves no contexto de *caches* WWW, onde visões parciais do conjunto de nós (um conjunto de caches) seria possível. Em um nível básico, uma função de *hash* consistente é tal que o seu resultado muda pouco à medida que a faixa da função (o conjunto de nós destino, por exemplo) se altera. Isso é exatamente o que buscamos para evitar os problemas de reconfiguração discutidos anteriormente.

O princípio básico é criar um espaço de endereçamento muito maior que o necessário para enumerar todos os elementos do conjunto a ser manipulado e atribuir a cada elemento uma chave aleatória nesse espaço. Ao invés de atribuir diretamente essas chaves a nós finais, estes últimos também recebem V identificadores aleatórios dentro do espaço. O conjunto de chaves assinalado a cada nó se torna uma função das posições relativas dos identificadores das chaves em relação aos identificadores dos nós — por exemplo, chaves podem ser assinaladas ao nó cujo identificador esteja mais próximo do identificador da chave dentro do espaço de endereçamento. Sendo assim, um novo nó incluído no sistema simplesmente determina seu(s) identificador(es) e se coloca dentro do espaço de endereçamento, se tornando o destino daquelas chaves próximas.

Um detalhe no caso de *hash* consistente inexistente no *hash* tradicional é que a tarefa de determinar o nó responsável por uma certa chave se torna mais complexa. Enquanto no *hash* tradicional a definição do nó de destino era feita simplesmente pela aplicação de uma função `mod` (ou `div`) ao valor da chave, no caso de *hash* consistente essa identificação é mais complexa. Supondo um sistema onde todos os nós têm conhecimento dos identificadores de todos os demais, pode-se definir um mapa de intervalos representando cada parte do espaço de endereçamento. Esse mapa pode ser implementado, por exemplo, com uma árvore balanceada. Usando-se uma estrutura desse tipo, determinar o nó responsável por uma chave se torna uma operação de complexidade $O(log(n))$, ao contrário do que ocorre com o *hash* tradicional, onde a operação é $O(1)$. Entretanto, isso pode ser melhorado para $O(1)$ na média, em troca de um espaço adicional da ordem de $O(n)$: ao invés de se construir uma árvore de intervalos, divide-se o espaço de endereçamento em n partes iguais, cada uma com sua árvore de intervalos. Determinar qual árvore deve ser consultada é $O(1)$, já que o espaço é dividido em partes iguais, e as árvores terão em média um nível apenas (já que há n intervalos). Para sistemas distribuídos com até alguns milhares de nós, o espaço adicional necessário é irrisório considerando-se as arquiteturas atuais.

A Fig. 3 ilustra o processo de distribuição e redistribuição com *hash* consistente (por simplicidade, um espaço linear é representado, mas um espaço circular é normalmente adotado na prática). Inicialmente, quatro identificadores de nós são gerados aleatoriamente no espaço, dividindo-o em quatro regiões. Um quinto nó é adicionado e recebe seu identificador aleatório, dividindo novamente o espaço entre os dois primeiros identificadores.

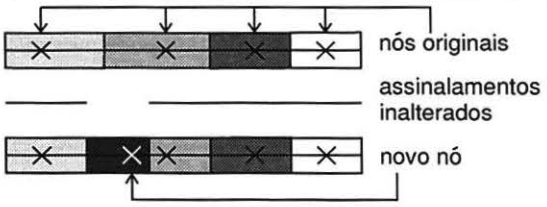

Figura 3. Redistribuição usando *hash* consistente

Como visto na figura, o volume de tráfego causado pela reconfiguração nesse caso se limita exclusivamente às chaves que precisam ser transferidas para o novo nó. Nenhuma troca de chaves ocorre entre nós que já existiam no sistema, o que reduz as comunicações ao mínimo, exatamente o comportamento que se procura para a reconfiguração com custo mínimo de comunicação entre nós.

Hash consistente possui entretanto mais um detalhe que precisa ser considerado: o conceito depende da utilização de identificadores aleatórios gerados para cada nó dentro do espaço de endereçamento. Isso significa que, ao contrário das soluções com *hash* tradicional, onde o particionamento do espaço podia ser feito com grande uniformidade entre os nós, o número de chaves associado a cada nó pode variar em função da distribuição obtida para as chaves aleatórias, como pode ser visto na figura. Se, por exemplo, os identificadores para n nós tiverem sido gerados de forma que a divisão do espaço fosse completamente uniforme, com k/n chaves para cada nó, a inclusão de um novo nó causaria um desbalanceamento entre as distribuições, pois o novo identificador se interporia entre entre dois identificadores já existentes, tornando-se responsável por metade do espaço de endereçamento existente entre os dois identificadores originais, que teriam seus espaços reduzidos em função da distância do novo identificador a cada um deles. Todos os

outros nós do sistema, entretanto, permaneceriam com k/n chaves, enquanto os três identificadores envolvidos teriam, em média, $2k/3n$ chaves cada.

Para reduzir esse problema, a proposta de *hash* consistente recomenda que cada nó do sistema na verdade gere um conjunto de V identificadores aleatórios. Dessa forma, ao invés de ter o impacto da inserção/retirada de um nó aplicado a apenas um par de nós vizinhos, esse impacto é distribuído entre V pares diferentes, reduzindo a variância resultante.

Uma variação do princípio de hash consistente considera o espaço de endereçamento como sendo circular e definindo a noção de proximidade como sendo direcional: um nó é responsável por todas as chaves que estejam a partir do seu identificador mas antes do identificador do próximo nó. As características do sistema se mantém praticamente inalteradas, exceto pelo fato de que nós que entram ou saem do sistema afetam apenas um vizinho, o que pode aumentar um pouco mais o desbalanceamento entre nós.

Considerando essas características, as seções a seguir apresentam a metodologia utilizada na análise do comportamento das diversas formas de distribuição de tarefas por funções de *hash* discutidas e os resultados dessa análise.

4. Metodologia

Neste trabalho decidimos avaliar o comportamento de quatro soluções para distribuição de tarefas por *hash*: as soluções tradicionais baseadas nas operações `mod` e `div` e uma aplicação de *hash* consistente variando o número de identificadores usados por cada nó dentro do espaço de endereçamento ($V = 1$ e $V = 10$). Dessa forma pudemos avaliar o impacto do aumento do número de identificadores, bem como o comportamento do sistema sem esse recurso, já que o mesmo possui menor complexidade.

Para determinar os padrões de geração de chaves pela aplicação, observamos aplicações reais atualmente sendo executadas dentro do nosso ambiente distribuído, o Formigueiro (*Anthill*). Em particular, observamos dois algoritmos de mineração de dados da plataforma Tamanduá[2], o *a priori*, para identificação de *itemsets* freqüentes usado para derivação de regras de associação [18], e o *k-means*, um algoritmo de classificação [7]. Ambos possuem como características comuns o fato de serem intensivos em processamento e entrada e saída de dados e por serem iterativos, fazendo uso pesado do recurso de rotulação de tarefas para garantir que novos dados gerados durante uma iteração sejam endereçados ao nó correto onde deverão ser processados e adicionados ao estado apropriado. Por outro lado, exceto pela natureza iterativa, os algoritmos têm estruturas bem diferentes, operam sobre dados que representam abstrações diferentes do domínio do problema e foram desenvolvidos por equipes diferentes.

Verificamos que em execuções usuais cada algoritmo gera entre 15.000 e 25.000 chaves diferentes. Uma característica até certo ponto inesperada é que em ambos os casos os programadores optaram por atribuir chaves simplesmente como inteiros seqüenciais, uma vez que só lhes interessava a identificação de cada tarefa como independente das demais. Decidimos então utilizar o mesmo artifício, considerando que essa solução gera as melhores distribuições possíveis para as soluções com *hash* tradicional. Para a geração de identificadores e para verificação do assinalamento de chaves a nós utilizamos a solução da rede *peer-to-peer* CHORD [16], que gera uma assinatura SHA1 para cada chave e para cada identificador de nó, mapeando valores de forma bem distribuída sobre um espaço de endereçamento de 160 bits. A implementação do hash consistente no CHORD utiliza a variação com espaço de endereçamento circular e direcionado discutida anteriormente. Essa versão será usada nos experimentos, por ser de implementação mais simples e por permitir avaliar o caso extremo de desbalanceamento no reassinalamento.

Para avaliar o desempenho das diversas soluções de distribuição de tarefas por *hash* consideramos um cenário simulado onde uma aplicação distribuída executa em um certo conjunto de nós n. Em um determinado momento, quando um certo número de tarefas já foi criado no sistema e distribuído entre os nós existentes, um novo nó é incorporado à aplicação, forçando a redistribuição das tarefas. Inicialmente, avaliamos a qualidade do balanceamento da distribuição de chaves entre os n nós obtida por cada tipo de *hash*. Depois, contabilizamos o número de tarefas que trocam de nó no momento da alteração do sistema. Computamos o número total de mensagens, bem como a variabilidade do padrão de comunicação, em termos do desvio padrão do número de tarefas que mudam de nós em relação à média a cada execução.

Cada configuração com *hash* tradicional foi executada apenas 1 vez, já que nesse caso, como as chaves são simplesmente seqüenciais, os resultados não variam. As simulações com *hash* consistente, dada a maior variabilidade das execuções, foram executadas trinta vezes para garantir valores convergentes. Em geral, cada simulação usou 15.000 chaves. Em alguns casos outros valores foram utilizados para verificar o comportamento do sistema quando o número de chaves cresce junto com a aplicação.

5. Análise dos resultados

Inicialmente, verificamos a variância na distribuição de chaves entre os nós do sistema à medida que o número de nós aumenta. Essa verificação serve para confirmar a

2 URL: `http://tamandua.speed.dcc.ufmg.br/`

correta aplicação do *hash* tradicional, que deve apresentar sempre baixa variância, bem como para avaliar o impacto do aumento do número de identificadores por nó no *hash* consistente (V), cujo objetivo é exatamente reduzir efeitos de uma redistribuição parcial das chaves. A figura 4 apresenta os resultados para as quatro técnicas relativos à média da distribuição (como o número de chaves foi mantido constante em 15.000, a média vai se reduzindo, na forma $15.000/n$).

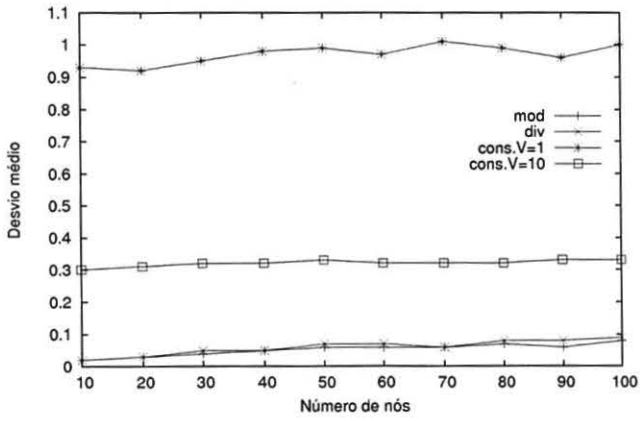

Figura 4. Desvio padrão da distribuição de chaves por nós (relativo à média)

Como se pode ver, a distribuição de chaves é bastante uniforme com as soluções de *hash* tradicionais, como seria de se esperar. O pequeno aumento da variância em ambos os casos (`mod` e `div`) se deve à redução do número de elementos por nó, quando a variação se torna mais sensível. É interessante observar que a variação da distribuição por *hash* consistente com $V = 10$, apesar de superior, ainda é aceitável. Como esperado, a variação com apenas um identificador por nó leva a distribuições de chaves sensivelmente desbalanceadas. Uma análise em separado das técnicas tradicionais (omitida por limitações de espaço) indica um comportamento mais uniforme para o *hash* por `mod`. A curva para o *hash* `div` apresenta pequenas irregularidades com a variação de n.

Medimos também o desvio padrão da distribuição de chaves após a inserção de um novo nó e a redistribuição das chaves. Os resultados são praticamente idênticos aos apresentados na figura 4, com um aumento pequeno, mas perceptível, do desvio relativo para a distribuição do *hash* consistente com um identificador por nó, confirmando o esperado, conforme discutido anteriormente.

O volume de comunicação pode ser verificado pela avaliação do número de tarefas que são reassinaladas por cada técnica, apresentado na figura 5. As curvas para as duas soluções com *hash* consistente são apresentadas novamente em separado, por não serem distingüíveis no primeiro gráfico.

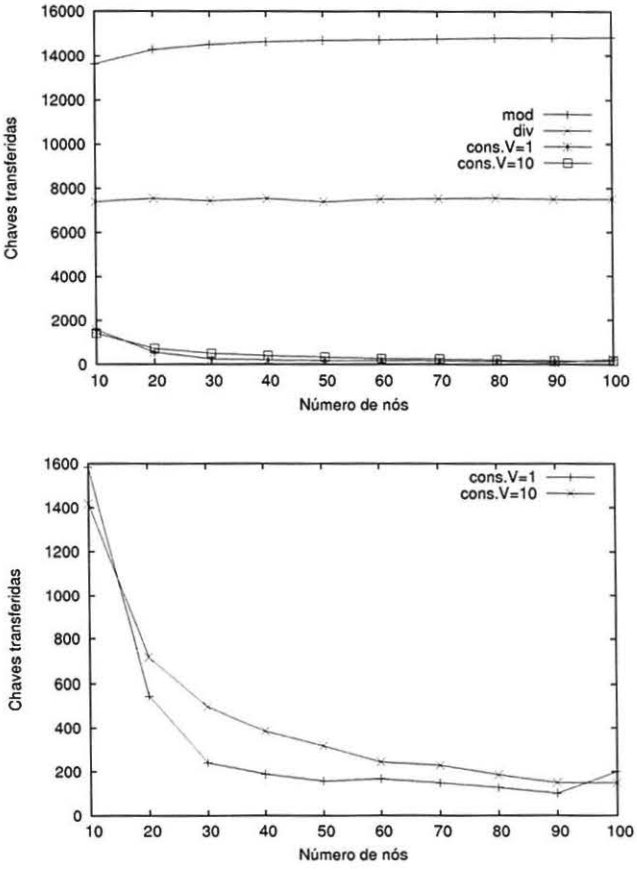

Figura 5. Número de tarefas reassinaladas

Os resultados confirmam a análise da Seção 3: o número de tarefas reassinaladas no *hash* `div` permanece praticamente constante por volta de 7.500 tarefas reassinaladas ($k/2$), independente do número de nós; já o número de mensagens do *hash* tradicional (`mod`) cresce ligeiramente, tendendo a um valor bem próximo a 15.000 tarefas. As soluções usando *hash* consistente atendem ao objetivo para o qual foram desenvolvidas: o número de tarefas reassinaladas é significativamente mais baixo que no caso do *hash* tradicional. Observando as curvas em detalhe, podemos ver que o comportamento com 10 identificadores por nó se comporta como esperado: o número de tarefas reassinala-

das decai com o número de nós, sempre próximo de k/n. A curva para o caso com apenas um identificador por nó já não é tão bem comportada. Como um nó que entra no sistema sempre recebe apenas parte das chaves de outro nó já no sistema (espaço circular direcionado), a transferência tende a ser em média inferior à esperada no caso ideal. Isso de certa forma é de se esperar, uma vez que o nó que entra no sistema nesse caso tende sempre a receber menos elementos que a média dos nós existentes.

Finalmente, sobre a transferência de tarefas, verificamos também o padrão de comunicação envolvido através da análise da variação no número de tarefas cedido por cada nó do sistema em relação à média de tarefas reassinaladas por nó (figura 6).

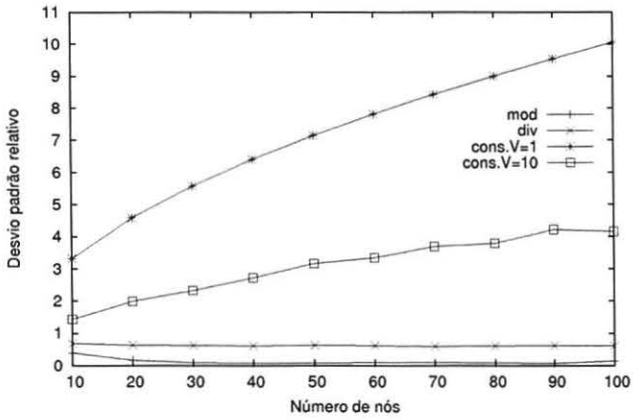

Figura 6. Variação no número de tarefas transferidas por nó em relação à média

Os resultados também confirmam o esperado: a variação no caso do *hash* tradicional com `mod` é bastante baixa, já que praticamente cada nó troca o mesmo número de tarefas com todos os demais. O padrão de comunicação no caso do *hash* com `div` também é bem definido, porém a variação é maior pela característica do método, onde cada nó troca um número de tarefas maior com o seu vizinho subseqüente. Entretanto, essa variação ainda é limitada e estável. Já nos mecanismos de *hash* consistente, o novo nó só recebe chaves/tarefas dos nós vizinhos aos identificadores adotados por ele. No caso simplificado ($V = 1$), o novo nó sempre recebe chaves apenas de um vizinho, logo o desvio padrão cresce rapidamente com o número de nós. Já no caso com diversos identificadores há mais nós envolvidos na transferência, mas ainda há um número crescente de nós que não transferem nenhuma tarefa, aumentando a variação no padrão de comunicação.

Como mencionado na Seção 4, realizamos também diversas simulações com um número maior de nós no sistema para avaliar situações de *scale-up*. Nesse caso, aumentamos o tamanho da aplicação (o número de chaves/tarefas geradas) na mesma escala em que aumentamos o número de nós no sistema, mantendo uma média de 1.500 chaves por nó de processamento (caso equivalente a 10 nós nos gráficos anteriores, que tinham 15.000 chaves no total). O comportamento do sistema se manteve sempre próximo ao apresentado nos resultados anteriores, indicando que as soluções mantém os mesmos padrões de comportamento com o crescimento do problema e do sistema. Por não acrescentarem novas informações, os gráficos não são apresentados.

6. Conclusões e trabalhos futuros

Neste trabalho apresentamos uma avaliação comparativa de quatro soluções de distribuição de tarefas baseadas no princípio geral de *hashing*. Duas formas de *hash* tradicional foram comparadas com o uso da técnica de *hash* consistente para o assinalamento de tarefas a nós em um sistema distribuído. Os resultados comprovam a maior estabilidade da distribuição baseada em *hash* tradicional, especialmente o *hash* baseado na função `mod`. Entretanto, o volume de tráfego devido à migração de tarefas quando de uma reconfiguração do sistema mostra que essa solução não é satisfatória em sistemas dinâmicos que admitem reconfigurações durante a execução. Nesse caso, as soluções baseadas em *hash* consistente se mostraram claramente superiores, em particular a solução utilizando mais de um identificador para cada nó participante do sistema. Nesse caso, a distribuição de chaves entre nós se mantém balanceada ao longo da execução, mesmo após inserções ou remoções. O nível de desbalanceamento, apesar de superior ao de soluções de *hash* tradicional, ainda é bastante aceitável. O tráfego de tarefas que são reassinaladas durante uma reconfiguração cai drasticamente quando comparado com soluções tradicionais, tendendo ao mínimo necessário para a transferência apenas das tarefas associadas ao nó que entra/deixa o sistema.

O principal caminho para extensão deste trabalho é sua aplicação a um sistema real. Uma solução baseada em *hash* consistente se encontra hoje em implantação no Formigueiro, um ambiente para execução de aplicações distribuídas desenvolvido no DCC-UFMG. Esse ambiente permite a execução eficiente de tarefas iterativas intensivas em processamento e entrada e saída em uma forma reconfigurável. Essa reconfiguração permitirá a inclusão de recursos de tolerância a falhas, escalonamento de tarefas dinâmico com base na demanda das aplicações e reconfiguração dinâmica do sistema para aten-

der a políticas de controle de recursos como energia e memória.

Outros trabalhos futuros incluem a avaliação de soluções baseadas em *hash* consistente que permitam o tratamento eficiente de questões sobre heterogeneidade[8]. Sistemas compostos por processadores com características diferentes podem se beneficiar de soluções que permitam a inclusão dessas diferenças no processo de distribuição de tarefas, atribuindo mais chaves a um processador mais poderoso, por exemplo. Outro fator importante é considerar o desbalanceamento de carga associado a diferenças de processamento entre tarefas. A solução usual com *hash* consistente assume que as tarefas tenham demandas semelhantes por recursos. É preciso avaliar soluções que permitam alterar o padrão de distribuição de tarefas caso algumas possam exigir mais recursos que outras, por exemplo.

Referências

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, nov. 2002.

[2] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[3] V. Cardellini, M. Colajanni, and P. S.Yu. Request redirection algorithms for distributed Web systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):355–368, abr. 2003.

[4] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in linda. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, págs. 236–242, New York, NY, USA, 1986. ACM Press.

[5] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, D. Paranhos, E. Santos-Neto, C. DeRose, T. Ferreto, M. Mowbray, R. Scheer, and J. Jornada. Scheduling in bag-of-task grids: The Pauá case. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2004)*. SBC, 2004.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2004.

[7] R. Fonseca, W. Meira Jr., D. Guedes, and L. Drummond. Anthill: A scalable run-time environment for data mining applications. In *Proceedings of the 17th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*. SBC, 2005.

[8] P. B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *Proceedings of the INFOCOM Conference*. IEEE, 2005.

[9] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*, págs. 654–663. ACM, 1997.

[10] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidinia, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the eighth international conference on World Wide Web*, págs. 1203–1213. Elsevier North-Holland, Inc., 1999.

[11] D. Lowenthal and G. Andrews. An adaptive approach to data placement. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*. IEEE, 1996.

[12] G. Manku. Routing networks for distributed hash tables. In *Proceedings of the 22nd ACM PODC*, 2003.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, págs. 161–172, New York, NY, USA, 2001. ACM Press.

[14] S. Ratnasamy, I. Stoica, and S. Shenker. Routing algorithms for DHTs: Some open questions. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, págs. 45–52, London, UK, 2002. Springer-Verlag.

[15] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, págs. 329–350, London, UK, 2001. Springer-Verlag.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, págs. 149–160, New York, NY, USA, 2001. ACM Press.

[17] G. Teodoro, T. Tavares, B. Coutinho, W. Meira Jr., and D. Guedes. Load balancing on stateful clustered web servers. In *Proceedings of the 15th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*. SBC, 2003.

[18] A. Veloso, W. Meira Jr., R. Ferreira, D. Guedes, and S. Parthasarathy. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, 2004.