Instituto de Matemática e Estatística
Universidade de São Paulo

Noções de escalonamento de tarefas via SLURM

Samuel Plaça de Paula

MAC5742 - Computação Paralela e Distribuída
Professor Alfredo Goldman vel Lejbman

São Paulo, junho de 2015

Sumário

1 Introdução

Observamos que nos últimos anos há uma tendência em computação, sobretudo a de alto desempenho, a buscar ganho de capacidade de processamento adicionando um número cada vez mais alto de processadores em paralelo [5]. Trata-se, em parte, de uma solução à inescapável limitação física ao aumento de *clock* dos processadores [7], que torna cada vez mais difícil seguir a Lei de Moore através das mesmas estratégias das últimas décadas [8].

No entanto, a adição de mais nós de computação, com cada vez mais recursos, como número CPUs, processadores com cada vez mais núcleos e *threads*, unidades CUDA, etc., não é trivialmente traduzida em maior desempenho. Aproveitar esses recursos da melhor forma possível é um desafio. Por exemplo, é preciso compreender quais tipos de tarefas e respectivas formas de implementação são mais adequadas a cada forma de organização das máquinas, desenvolver software atendendo a tais especificidades, entre outros problemas que cabem aos usuários dessas máquinas.

Outra tarefa que em pequena escala poderia ser atribuída a um ser humano é escolher de quais recursos cada tarefa a ser executada irá dispor. No entanto, quando estamos pensando em máquinas como as do Top 500, com centenas de milhares de núcleos de processamento distribuídos em topologias sofisticadas [5], e que provavelmente irão se encarregar de computações divididas em um número ainda maior de pequenas tarefas, não é razoável que usuários se encarreguem de decidir quando e onde essas tarefas serão executadas.

Mas estamos falando de um contexto em que, para cada computação, praticamente todas as tarefas geradas são do tipo *batch*, isto é, do tipo que não requer interação com o usuário. Portanto, a execução dessas tarefas pode ser gerenciada por um software, que é o *escalonador de processos*. Ele deve atribuir a cada tarefa determinados recursos computacionais e uma ordem de disparo, de acordo com determinadas especificações, como:

- há tarefas que dependem do resultado final de outras e portanto devem respeitar uma ordem de disparo para evitar que ocupem recursos enquanto realizam uma espera;
- há tarefas que precisam de comunicação entre si, de modo que é importante que elas estejam alocadas em recursos próximos na topologia, a fim de que a troca de mensagens seja rápida e não congestione os canais de comunicação;
- desejamos que a todo momento esteja em uso o máximo possível de recursos computacionais;

entre outros. Tais requisitos preocupam-se tanto com a velocidade do término da computação quanto com questões como uso de energia, evitar desperdício dos recursos, etc. Note que, como o escalonador de processos é um programa a ser executado na própria

máquina cujos recursos estão sendo administrados, ele mesmo deve poder ser executado de forma eficiente. Assim, como voltaremos a discutir, existe um equilíbrio entre o quanto queremos que o escalonador tome boas decisões e quanto tempo e recursos computacionais gostaríamos que essas tomadas de decisões consumissem.

O presente texto se propõe a dar uma visão bastante básica sobre como funciona um escalonador de tarefas, especialmente no contexto de computação de alto desempenho (ou HPC, *High Performance Computing*). Para tanto, apresentamos alguns conceitos básicos de escalonamento, baseados no livro de Maciej Drozdowski [1] e exploramos em especial o escalonador SLURM, um escalonador de tarefas usado de forma expressiva em HPC – por exemplo, em 2014, foi citado como o gerenciador de recursos padrão em 5 dos 10 computadores mais rápidos do TOP500 [3].

Discutiremos um pouco uma das principais estratégias do SLURM para escolha dos recursos alocados, que é o aproveitamento da localidade dos recursos na topologia. Em particular, mostraremos de forma básica como isso é feito em topologias do tipo grade (2D ou 3D). Para tanto, precisaremos de alguns conceitos de otimização combinatória, que é a linguagem usada para modelar esse problema. Tais conceitos também estão contidos de forma simplificada no presente texto.

2 Conceitos básicos de escalonamento

Como já mencionado, iremos nos basear, aqui, nos conceitos como definidos no livro de Drozdowski [1]. Algumas das noções às quais nos referimos anteriormente serão definidas de maneira um pouco mais precisa. Além disso, na medida em que acharmos possível e adequado, daremos o equivalente de cada conceito no caso do SLURM, que é o escalonador de processos que discutiremos aqui.

Chamaremos de *tarefa* (ou *task*) uma entidade escalonável atômica. Isto é, uma tarefa será aquilo que definirmos como a menor unidade possível de computação em um determinado contexto. Pode ser uma tarefa, um processo, um conjunto de processos, uma comunicação, etc., conforme o computador e o modelo de computação que estamos levando em conta. No contexto do SLURM, em geral poderemos pensar em cada tarefa como uma thread.

Um conjunto de tarefas pode ser parcialmente ordenado por uma relação de dependência, em que uma determinada tarefa só pode começar a ser executada após o término de alguma(s) outra(s). Podemos representar essas dependências através de um grafo dirigido acíclico (DAG). Um conjunto de tarefas formando um sistema de dependências representados por um DAG é chamado de *job*.

Definimos um *processador* ou *elemento de processamento* é a menor entidade ativa capaz de executar tarefas. Consideramos que, a qualquer momento, um processador só

pode ter, no máximo, uma tarefa ocupando-o. Portanto, o que definimos como processador também pode variar em granularidade conforme o caso: podem ser threads de CPU, núcleos (cores) de CPU, cada CPU inteira, um computador todo, etc. Embora o SLURM permita ativar o uso de diferentes threads dentro de um mesmo núcleo, a granularidade de processador considerada por seus algoritmos é a de um núcleo [9].

Como temos um grande número de processadores, é preciso levar em conta a forma como eles estão organizados para escalonar de forma eficiente. Podemos representar essa organização através de um grafo em que os vértices representam processadores e há uma aresta entre dois processadores se estes possuem um canal de comunicação mútua. Esse grafo é chamado de *grafo de hosts*, e a organização que ele representa é a *topologia* da máquina. O suporte físico à comunicação entre processadores é chamado de *rede de interconexão*, ou simplesmente de *rede*.

Chamamos genericamente de *recursos* todas as entidades de que as tarefas precisam para sua execução, como memória e banda de comunicação. Diferentemente de nosso texto de referência, aqui também chamaremos os processadores de recursos, pois essa diferenciação não será importante no nível de profundidade desse trabalho.

Um *escalonamento* é uma atribuição das tarefas a processadores e recursos ao longo do tempo. Um *escalonador* (ou *scheduler*) é um agente de software cujo trabalho é determinar um escalonamento conforme são submetidas a ele tasks, jobs, descrições dos processadores e recursos disponíveis, etc. No caso do SLURM, o uso comum é começar a execução submetendo ao sistema uma descrição das características das tarefas e dos processadores: são descritas dependências entre as tarefas, número de processadores a serem ocupados por cada tarefa, número de processadores a serem usados em cada CPU ou computador, etc. No entanto, durante a execução, o usuário pode lançar novas tasks, pausar ou cancelar algumas das que estão na fila (ou sendo executadas), entre outros recursos interativos.

3 Otimização combinatória e algoritmos de aproximação

A área de otimização combinatória se presta à descrição, estudo de propriedades e resolução algorítmica de problemas em que buscamos, de modo geral, subestruturas ótimas em uma estrutura finita que pode ser representada computacionalmente. Devido à natureza finita desses problemas, em geral é imediato que existe uma maneira de resolvê-los em tempo finito. No entanto, a dificuldade se dá quando consideramos a necessidade de encontrar algoritmos eficientes, principalmente pensando em consumo de tempo. Assim, é necessário encontrar maneiras inteligentes de explorar os espaços de soluções que podem ser muito grandes – digamos, de tamanho exponencial no tamanho da descrição da entrada do problema.

Há uma quantidade imensa de problemas práticos que podem ser modelados como

problemas de otimização combinatória. Por exemplo, encontrar um conjunto mínimo de pedaços suspeitos de código de forma que, com os pedaços tomados, seja possível determinar "assinaturas" de todo um conjunto dos vírus de computador conhecidos. Ou ainda, determinar onde devem ser abertos centros de distribuição de um certo produto de modo que nenhuma loja que o vende esteja muito distante do centro de distribuição mais próximo.

Porém, muitas vezes, as modelagens mais úteis dos problemas que desejamos resolver acabam tendo a propriedade de que é muito difícil resolvê-las computacionalmente. Essa "dificuldade" tem um sentido matematicamente bem-definido que iremos explicar de forma simplificada nessa seção. Quando nos deparamos com esses problemas difíceis e queremos resolvê-los rapidamente, muitas vezes temos que abrir mão de resolvê-los exatamente, chegando apenas a soluções aproximadas, que em muitos casos são suficientemente boas para se trabalhar. Os métodos que temos para atingir tais soluções aproximadas são chamados *algoritmos de aproximação*, conceito que também veremos adiante.

Como comentamos anteriormente, muitos dos modelos usados para estudar e descrever computação paralela usam os conceitos e a linguagem da área de otimização combinatória. Portanto, a presente seção se presta a apresentar alguns desses conceitos. Particularmente, procuramos descrever o suficiente para que seja possível referenciar no restante do texto.

3.1 Problemas de otimização combinatória

Definimos um problema de otimização como um conjunto de instâncias. Para cada instância I, temos um conjunto de *soluções viáveis*, $\mathrm{Sol}(I)$. A cada solução viável $S \in \mathrm{Sol}(I)$, está associado um valor numérico, $\mathrm{val}(S)$.

Dada uma instância I, desejamos encontrar $S^* \in \mathrm{Sol}(I)$ tal que $\mathrm{val}(S^*)$ seja *mínimo*, para problemas de minimização, ou *máximo*, para problemas de maximização. Um tal S^* é chamado de *solução ótima*, e seu valor $\mathrm{val}(S^*)$ é o *valor ótimo*, denotado por $\mathrm{OPT}(I)$, ou simplesmente OPT, quando a instância está subentendida.

Se $\mathrm{Sol}(I) = \emptyset$, dizemos que a instância I é *inviável*.

Vamos exemplificar a definição acima com um problema clássico, o *problema do emparelhamento máximo*. Uma instância desse problema é constituída por:

- uma lista $a_1, a_2, \ldots, a_n$ de *operários*;
- uma lista $m_1, m_2, \ldots, m_p$ de *máquinas*; e
- para cada operário a_i, um conjunto $M_i \subseteq \{m_1, \ldots, m_p\}$ representando as máquinas que o operário i é capaz de manejar.

Uma solução viável S dessa instância é um conjunto não vazio de pares ordenados

$$S = \{(a_{i_1}, m_{i_1}), \ldots, (a_{i_k}, m_{i_k})\},$$

onde cada operário a_i e cada máquina m_j aparece no máximo em um dos pares, e para todo ℓ vale que $m_{i_\ell} \in M_\ell$. Isto é, cada par é uma atribuição de um operário a uma máquina que ele é capaz de operar. Uma solução viável S é chamada de *emparelhamento*, e atribui no máximo uma máquina a cada operário, e no máximo um operário a cada máquina. Note que uma instância desse problema só é inviável se nenhum operário é capaz de manejar máquina alguma. Dessa forma, não existirá nenhum conjunto não vazio de pares que seja viável.

Considere uma solução viável S. Definimos o valor desse emparelhamento como sendo a sua cardinalidade $|S|$, ou seja, o número de pares que a solução contém. Desejamos encontrar um emparelhamento de cardinalidade máxima. Daí o nome do problema: um emparelhamento S^* cuja cardinalidade é máxima é chamado de *emparelhamento máximo*. Podem existir vários emparelhamentos máximos distintos para uma mesma instância.

Note que o número de soluções viáveis pode ser muito grande em relação ao tamanho da instância. Mais especificamente, pode ser exponencialmente grande. Assim, a abordagem mais simples possível de enumeração direta de soluções pode ser computacionalmente muito custosa. Felizmente, no entanto, esse é um problema que podemos resolver de forma eficiente. Nesse contexto, isso quer dizer que existe um algoritmo que, para qualquer instância, encontra uma solução ótima e é garantido que o algoritmo nunca terá que examinar explicitamente uma porção grande do espaço de soluções. Vamos definir essa noção de moda mais precisa a seguir.

Dizemos que um problema de otimização combinatória *admite uma solução eficiente*, ou *de tempo polinomial*, se existe um algoritmo A tal que, para qualquer instância I do problema, ao executarmos A com a representação computacional $\langle I \rangle$ da instância, o algoritmo irá terminar após um número de passos polinomial no tamanho de $\langle I \rangle$ e escreve na saída uma representação $\langle S^* \rangle$ de uma solução ótima S^* para a instância I.

Existe um conjunto de problemas computacionais que dizemos formar a chamada classe de complexidade NP. Dentre esses, existem alguns que admitem solução eficiente, que formam a classe P. Existem alguns para os quais não conhecemos solução de tempo polinomial, e que possuem a seguinte propriedade: a existência de solução eficiente para qualquer um deles implica a existência de solução eficiente para todos os outros, e que assim P = NP. São os problemas chamados NP-completos.

Existe uma classe ainda mais abrangente, dos problemas NP-difíceis. Chamamos um problema de NP-difícil se ele tem a propriedade de que, caso exista para ele uma solução eficiente, então P = NP. Não sabemos se essa igualdade é verdadeira e, de fato, trata-se provavelmente do maior problema em aberto em computação.

Em termos informais, podemos dizer que resolver eficientemente um problema NP-difícil equivale a resolver eficientemente todo um conjunto de problemas que estão sem

solução polinomial conhecida possivelmente há décadas. Isso não significa que seja impossível (simplesmente não sabemos), mas que tentar resolver de forma exata qualquer um desses problemas é um desafio muito maior do que pode parecer a princípio.

Como mencionamos anteriormente, muitos dos problemas mais interessantes de otimização combinatória são NP-difíceis. Assim, quando nos deparamos com um deles, em geral precisamos obter um algoritmo que abra mão de pelo menos uma das seguintes propriedades:

- Funcionar para qualquer instância;
- Obter respostas eficientemente;
- Obter respostas exatas.

A primeira opção corresponde a obter algoritmos especializados a certas famílias "fáceis" de instâncias. A segunda opção é aquela feita por *solvers* de programação linear inteira, por exemplo. Eles sempre garantem achar uma resposta ótima, e fazem diversas otimizações nos algoritmos empregados, mas não são capazes de garantir eficiência *sempre*.

A terceira opção corresponde a abrir mão da garantia de respostas ótimas. No entanto, podemos fazer isso de forma "controlada" através de algoritmos de aproximação. Considere um problema de otimização combinatória. Um *algoritmo de aproximação* para esse problema é um algoritmo que, para qualquer instância I do problema, termina sua execução após um número de passos polinomial no tamanho da representação computacional $\langle I \rangle$ da instância, e encontra uma solução viável S' (se alguma existir), com a propriedade de que existe um fator α tal que $\text{val}\,(S') \leq \alpha \cdot \text{OPT}(I)$. Esse fator α pode ser uma constante, uma função de alguma característica da instância, etc., e é chamada de *garantia de desempenho*.

3.2 O problema *Bin Packing online*

O problema *Bin Packing* é o problema em que, dados vários objetos cujos tamanhos são informados em uma certa ordem, desejamos encontrar uma forma de empacotar esses objetos no menor número possível de cestos, sendo que todos os cestos são iguais, isto é, têm a mesma capacidade. Os objetos precisam ser alocados na ordem em que seus tamanhos são fornecidos, de modo que essa é a chamada variante *online* do Bin Packing. Isso significa que não podemos ordenar os processor por tamanho antes de empacotá-los, por exemplo.

Podemos defini-lo formalmente da seguinte maneira. Uma instância I do Bin Packing consiste de uma sequência (e não simplesmente um conjunto, pois a ordem é importante)

$$L = (a_1, a_2, \ldots, a_n)$$

de *tamanhos de itens*: cada item i tem um tamanho a_i, que é um número racional em $(0, 1]$. Desejamos encontrar o mínimo inteiro m e uma partição dos itens em conjuntos, ou *bins*, $B_1, \ldots, B_m$, onde $\sum_{j \in B_i} a_j \leq 1$. Isto é, cada B_i é um *cesto* (bin) com tamanho ocupado no máximo 1.

Note que a capacidade de cada cesto ser 1 é suficientemente genérica; se todos os cestos têm uma capacidade V arbitrária, basta escalar os tamanhos dos objetos.

Esse problema, a despeito de sua aparente simplicidade, é NP-difícil [4]. No entanto, como veremos, ele é uma forma útil de modelar o problema de designar tarefas a processadores em uma topologia unidimensional. (Para tanto, iremos usar técnicas que transformam as topologias reais das máquinas em equivalentes 1D suficientemente bons.)

Essa utilidade vem do fato de que existem boas aproximações (algoritmos com boas garantias de desempenho) que são bastante rápidas. Isto é, um escalonador de processos pode se dar ao luxo de executar um algoritmo desses e assim resolver rapidamente instâncias do problema. A importância da eficiência do algoritmo vem do fato de que um escalonador não pode demorar para tomar suas decisões, uma vez que o objetivo é justamente otimizar o uso de recursos.

Citamos aqui o algoritmo Best Fit (BF). Este algoritmo segue a seguinte regra: a cada objeto k processado, já temos alguns *bins* abertos, e nos quais estão armazenados alguns dos objetos. Se existir um bin B_i aberto tal que $\sum_{j \in B_i} a_j \leq 1 - a_k$, colocamos o objeto k em um tal bin dando preferência ao bin que ficará mais cheio possível após essa alocação. Isto é, colocamos k no bin que ficará com capacidade mais próxima de (mas menor que) 1 como resultado dessa alocação. Se não houver nenhum bin com capacidade restante suficiente para o objeto k, então abre-se um novo bin e coloca-se k nele.

Esse algoritmo tem uma implementação ingênua quadrática: a cada objeto processado, percorre-se todos os bins abertos verificando se existe algum que ainda comporta aquele objeto, e qual é o melhor segundo o critério Best Fit. No entanto, existem implementações mais sofisticadas que exigem tempo $O(n \log n)$ [4], onde, relembramos, n é o número de objetos.

De modo informal, isso significa que, ao final da execução, a alocação de n processos terá tido custo sublinear para cada processo que chegou para o escalonador.

Além disso, o Best Fit é uma aproximação com garantia de desempenho 17/10 para o Bin Packing online. Note que é uma razão inferior a 2, e considerando que é o pior caso, podemos esperar que em geral o BF gere boas soluções. De fato, Leung et. al. mostram um determinado caso de estudo em que uma regra de escalonamento baseada em BF deu os melhores resultados práticos dentre as regras que esses autores consideraram suficientemente rápidas para valer a pena considerar [6].

4 Uma visão geral sobre o SLURM

SLURM, ou *Simple Linux Utility for Resource Management* ("utilitário simples para gerenciamento de recursos em Linux", em tradução livre), é um sistema de software livre, sob licença GNU GPLv2, para escalonamento de tarefas. Hoje em dia é comum ver seu nome grafado simplesmente como Slurm, em minúsculas e sem expansão do acrônimo, possivelmente em decorrência do fato de que atualmente ele pode ser instalado em diversos sistemas operacionais modernos baseados em Unix, como a família BSD, Mac OS X e Solaris, e não requer modificações no kernel para sua instalação. É desenvolvido em linguagem C e conta com mais de 500 mil linhas de código. Nessa seção descreveremos um pouco de sua história e funcionamento, e a maioria das informações está baseada na página oficial mantida pela SchedMD [9].

O Slurm começou a ser desenvolvido em 2002 como colaboração entre o Lawrence Livermore National Laboratory, a Linux NetworX, a HP e o Groupe Bull. Sua principal motivação foi estabelecer a existência de um bom gerenciador de recursos livre e com desenvolvimento suportado primariamente por entidades interessadas em computação de alto desempenho.

Desde então, outras entidades passaram a apoiar o projeto, dentre elas CEA, Cray, Intel e NVIDIA. Note-se que o Slurm possibilita também gerenciar os recursos CUDA, essenciais ao poder computacional de vários computadores atuais do TOP500 [5]. E, como já citamos anteriormente, houve uma adesão expressiva ao Slurm, visto que é o scheduler padrão de pelo menos 5 dos 10 computadores mais rápidos do TOP500 [3], dentre os quais o atual primeiro colocado, Tianhe-2, além do IBM Sequoia.

Em 2010, Morris Jette e James D. Auble deixaram o Lawrence Livermore National Laboratory para fundar a SchedMD. Atualmente, essa empresa é a principal mantenedora do projeto Slurm, além de oferecer serviços como treinamento e consultoria. A SchedMD mantém a atual página oficial do Slurm, que conta com material gratuito, em texto e vídeo, de suporte a usuários e desenvolvedores do software.

4.1 Recursos disponíveis e arquitetura básica

A função central do Slurm é atuar como escalonador de tarefas, designando as tarefas que são passadas a ele a determinados recursos computacionais ao longo do tempo. O sistema também oferece ao usuário uma interface para interação em tempo real, para iniciar, terminar e monitorar tarefas. Ainda, o Slurm também gerencia contenção de recursos mantendo uma lista de tarefas pendentes. Como já mencionado na Seção 2, portanto, o sistema tipicamente recebe requisições de um grande número de tarefas, com especificações quanto aos recursos que se permite usar, interdependências, etc., e também lida com alterações na carga em tempo real. O Slurm ainda possui diversas funcionalidades que podem ser instaladas opcionalmente, como limitação de recursos por usuário ou

conta bancária associada, seleção de recursos de forma otimizada a determinadas topologias (como veremos adiante) e ativação de preempção de acordo com grupos e prioridades de tasks.

O Slurm tem um gerenciador centralizado, o daemon `slurmctld`, que monitora os recursos e as tarefas em execução ou pendentes. Em geral há mais de uma instância desse daemon como *backup*, para fins de tolerância a falha. Também há, opcionalmente, um daemon `slurmdbd`, que se encarrega da comunicação com a base de dados, se houver. Essa base de dados pode ser compartilhada por vários clusters, e portanto por várias instâncias diferentes do Slurm, como se pode observar na Figura 4.1.

Cada nó de computação possui um daemon `slurmd` que fica esperando trabalho do daemon gerenciador. Ao receber trabalho, executa, devolve o status e volta ao estado de espera. De acordo com a documentação oficial, esses daemons formam uma estrutura hierárquica dentro da topologia do cluster, e com replicação para tolerância a falhas.

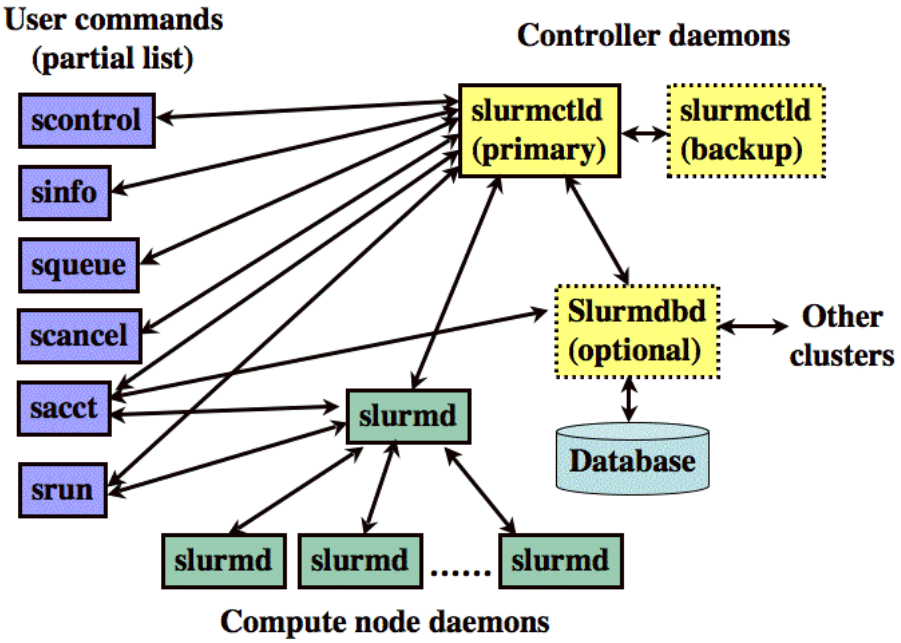

Figura 4.1: Arquitetura do Slurm, figura encontrada na página oficial mantida pela SchedMD [9]. Os componentes amarelos são daemons de controle geral, os verdes são daemons de controle de execução em cada computador, e os azuis são alguns dos serviços disponíveis para o usuário.

Finalmente, há as ferramentas para o usuário, como o `srun`, que pode ser usado para iniciar jobs, `scancel` para terminar jobs em execução ou na fila, `sinfo` para obter um relatório do estado do sistema e `squeue` para verificar o estado dos jobs. Os comandos `smap` e `sview` geral relatórios gráficos sobre o estado dos recursos e dos jobs, incluindo visualização da topologia da rede. O comando `scontrol` pode ser usado para monitorar

ou modificar as configurações e informações sobre o cluster. Todas as funções fornecidas pelo Slurm contam com APIs.

Note que o termo que usamos para nos referir aos recursos disponíveis ao usuário para gerenciar a carga do sistema, usamos o termo "job". De fato, estamos usando a palavra no mesmo sentido definido na Seção 2: um conjunto de tarefas, possivelmente paralelizáveis (mas possivelmente com algumas dependências que tornam algumas delas "seriais entre si"). Um conjunto de tarefas pertencentes a um job é atribuído a uma *partição* (*partition*), que é como o Slurm se refere a um conjunto de processadores. Como já discutimos, é desejável que, portanto, cada partição seja um conjunto de processadores próximos na topologia, para que as tarefas de um mesmo job possam se comunicar com o menor sobrecusto possível para a rede, assim economizando tempo não só para esse job como possivelmente para todo o cluster. Veremos uma descrição simplificada de como isso pode ser atingido na próxima seção.

5 Alocação de processadores por localidade

Nessa seção, iremos discutir brevemente uma estratégia que pode ser adotada para alocação de processadores obedecendo preferência por manter localidade. Trata-se do uso de curvas de Hilbert para reduzir o problema de alocação em topologias do tipo grade (*mesh*) 2D ou 3D, ou toro (*torus*) 2D ou 3D a um problema de alocação unidimensional que pode ser modelada como o problema Bin Packing online que vimos na Seção 3.2.

Esse é um dos métodos utilizados pelo Slurm, e é o padrão para as referidas topologias, bastante populares no TOP500. O Slurm também oferece um algoritmo mais adequado a topologias do tipo *fat tree*, também populares entre os computadores mais poderosos. Além disso, observamos que obter localidade através do emprego de estratégias de alocação unidimensional a topologias multidimensionais via transformações "bem escolhidas" é uma estratégia conhecida e empregada não apenas pelo Slurm [1, 6].

5.1 Descrição em alto nível do algoritmo

A ideia básica é a seguinte. Suponha que estamos considerando como unidades de processamento cada computador (nó) de um cluster, e que estes têm a mesma capacidade computacional e estão organizados em uma topologia unidimensional (digamos, em linha). Suponha que cada job será alocado a uma partição desses nós. Cada job é lançado com uma especificação de quantos processadores ele precisa. Isso pode ser definido pelo usuário ou detectado automaticamente. Podemos tomar como quantidade de processadores usados por um job o número máximo de tasks que podem ser executadas em paralelo dentro desse job, por exemplo.

Nesse cenário, podemos considerar que o problema de alocação de processos pode ser interpretado como o Bin Packing online. Cada bin será um nó (computador), e suas capacidade é o número de núcleos (cores) de processador que esse computador tem. Como estamos supondo que os nós são idênticos, os bins têm a mesma capacidade. O tamanho de um job é o número de processadores que ele requer. Alocar os objetos à menor quantidade possível de bins equivale, então, a alocar jobs ao menor número possível de nós diferentes. O resultado é que, assim, o conjunto de nós pelo qual cada job está espalhado se torna o mais compacto possível, e assim as tarefas dentro do job estão em localidades razoavelmente contíguas na topologia.

O que descrevemos agora deixa três problemas a serem resolvidos. O primeiro é que o Bin Packing online é NP-difícil. No entanto, já vimos na Seção 3.2 que esse problema admite uma boa aproximação que pode ser implementada de forma bastante eficiente. O segundo é que, diferente do Bin Packing, no nosso problema o número de bins a serem abertos é finito. Ou seja, pode ocorrer de precisarmos alocar mais um job e ele não poder pegar uma partição nova para si. Nesse caso, podemos espalhar as tarefas desse job por diferentes bins (nós) já alocados, procurando minimizar a distância entre o primeiro e o último nó utilizados.

Finalmente, resta resolver o problema de que para chegar até aqui fizemos duas hipóteses simplificadoras. Uma é que os nós são idênticos, para poder modelar o problema como Bin Packing. Essa hipótese é razoável, pois é bastante comum que os clusters sejam formados por uma grande quantidade de nós iguais [1, 5]. A outra hipótese é a de que usar poucos bins se traduz em partições de nós contíguas na topologia. Isso é válido se estamos alocando os bins em ordem numa topologia unidimensional. Mas essa hipótese não é realista! No entanto, podemos usar uma correspondência entre a topologia real e a topologia em linha. Se nossa correspondência tiver a propriedade de que posições vizinhas na linha representam posições próximas na topologia real, finalmente teremos boas condições de aplicar nossa ideia.

O exemplo que daremos aqui, como já dissemos, é adequado a topologias do tipo grade (2D ou 3D) ou toro (2D ou 3D). Podemos pensar que, nessas topologias, cada nó é um ponto no plano ou no espaço. Uma correspondência como a que estamos buscando seria uma curva unidimensional que passe por todos os pontos de um pedaço do plano ou do espaço. Diversas curvas têm essa propriedade, e aqui usaremos as *curvas de Hilbert*.

Podemos exemplificar para o caso bidimensional, como na Figura 5.1. A curva é uma construção fractal que no limite passa por todos os pontos da porção do plano em que está contida. Para nós, basta usar tantas iterações quantas necessárias para que a curva permita diferenciar as posições dos nossos nós. A curva tem que pontos extremos, de modo que podemos orientar nossa correspondência começando por qualquer um deles e avançando rumo ao outro. A numeração dos nós é feita de acordo com a ordem dada pela orientação que escolhemos para a curva. Ao fim do processo, toda a grade está numerada. Note que o mesmo processo serve para um toro 2D. Como existem curvas que preenchem o espaço para qualquer dimensão finita, essa técnica também pode ser empregada a topologias tridimensionais.

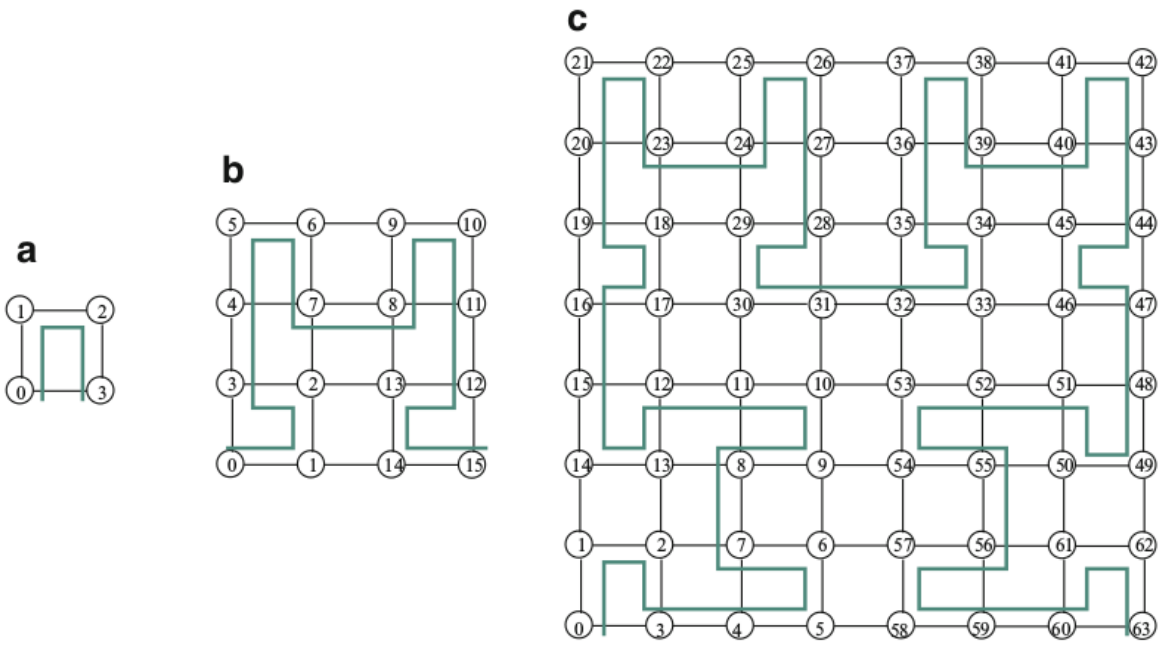

Figura 5.1: Exemplo de três iterações da curva de Hilbert no plano. Figura devida a Drozdowski, 2009 [1]. Os nós da topologia são numerados de acordo com a ordem dada pela curva.

Assim, podemos resumir a estratégia (em alto nível) da seguinte forma:

1. Use uma curva que preenche o espaço (como curvas de Hilbert) para obter uma correspondência "boa" entre a topologia real e uma topologia unidimensional.
2. Reduza o problema de alocação de processadores ao Bin Packing online, aplicando uma boa estratégia para esse problema a fim de resolver a alocação de forma rápida e obtendo uma boa solução.

Esse tipo de estratégia foi descrito em detalhes por Leung et. al. [6]. Nos testes realizados pelos autores sobre topologias de grade 2D e 3D, o uso dessa estratégia foi efetiva no sentido de que levou a menor tempo de término total das tarefas. O melhor tempo de makespan, e com menor desvio-padrão, foi atingido quando o algoritmo usado para resolver o passo 2 foi o Best Fit, conforme havíamos adiantado na Seção 3.2.

6 Conclusões

Ao estudar o funcionamento e utilização do Slurm, bem como os conceitos envolvidos em escalonamento de processos, observamos se tratar de uma área cada vez mais crucial para

o bom aproveitamento de recursos. Tanto no sentido computacional quanto energético, é um desafio cada vez maior obter uma bom aproveitamento e eficiência. Conforme mais atividades passam a se valer de alto poder computacional, e o ganho de poder computacional passa a ser buscado num paradigma de alta paralelização, a impressão que temos é que o escalonamento se torna um processo mais difícil e, na mesma medida, mais crucial. Somando-se a isso os fatos de que esse é um poder computacional também muitas vezes controlado por agentes de peso, como governos e grandes corporações, e de que a atual conjuntura mundial pressiona por um aumento urgente na eficiência energética de nossas atividades, o peso que se põe sobre esse problema computacional é maior ainda.

Consideramos bastante interessante estudar um pouco mais sobre os conceitos de escalonamento, e como essa área aproveita a linguagem e os resultados de uma área teórica de computação, a otimização combinatória. Vimos, através do Slurm, que algumas das estratégias vindas dessa área podem ser aplicadas de forma bastante direta e, aparentemente, efetiva, visto que o software em questão é bastante popular entre os computadores do TOP500.

Notamos, ainda, que a quantidade de conceitos envolvidos para se entender melhor essa comunicação entre as áreas é grande. Embora não sejam ideias necessariamente complexas, há uma grande variedade delas, e isso requer uma conciliação de diversas formas de pensar. No entanto, nos parece que as soluções obtidas dessa junção de disciplinas ainda são relativamente simples, como a vista na Seção 5. Gostaríamos de investigar se há maneiras de aproveitar ainda melhor esses diferentes conhecimentos acumulados para desenvolver estratégias ainda mais fortes de escalonamento, visto que é uma aplicação com importância crescente.

Referências

[1] Maciej Drozdowski. *Scheduling for Parallel Processing*. Springer, Dordrecht, 2009 edition edition, September 2009.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York u.a, first edition edition, January 1979.

[3] Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy Accounting and Control with SLURM Resource and Job Management System. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, number 8314 in Lecture Notes in Computer Science, pages 96–118. Springer Berlin Heidelberg, 2014.

[4] Dorit Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. Course Technology, Boston, 1 edition edition, July 1996.

[5] Volodymyr Kindratenko and Pedro Trancoso. Trends in High-Performance Computing. *Computing in Science & Engineering*, 13(3):92–95, May 2011.

[6] V.J. Leung, E.M. Arkin, M.A. Bender, D. Bunde, J. Johnston, Alok Lal, J.S.B. Mitchell, C. Phillips, and S.S. Seiden. Processor allocation on Cplant: achieving general processor locality using one-dimensional allocation strategies. In *2002 IEEE International Conference on Cluster Computing, 2002. Proceedings*, pages 296–304, 2002.

[7] Lev B. Levitin and Tommaso Toffoli. The fundamental limit on the rate of quantum dynamics: the unified bound is tight. *Physical Review Letters*, 103(16), October 2009. arXiv: 0905.3417.

[8] Robert R. Schaller. Moore's Law: Past, Present, and Future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[9] SchedMD. Slurm Overview. http://slurm.schedmd.com/overview.html, acessado 24/06/2015.

[10] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, 1 edition edition, April 2011.