6
Metaheurísticas Paralelas com Execução Autonômica em Grids

Este capítulo concentra as principais contribuições desta tese: uma nova estratégia de paralelização para metaheurísticas executarem em ambiente *grid*, um *middleware* de gerenciamento capaz de controlar a execução da aplicação no ambiente e a integração entre o *middleware* e a estratégia, a fim de tornar as metaheurísticas paralelas aplicações autonômicas, capazes de se auto-adaptarem às mudanças do ambiente. Inicialmente, na Seção 6.2 são apresentados os detalhes específicos da estratégia de paralelização proposta. Em seguida, as políticas de gerência necessárias para a utilização da estratégia são discutidas na Seção 6.3. O *middleware* usado para gerenciar o ambiente e tornar as implementações autonômicas é descrito na Seção 6.4. Para finalizar, as Seções 6.5.1 e 6.5.2 descrevem as duas heurísticas paralelas desenvolvidas neste trabalhado para validar as idéias propostas.

6.1
Considerações Iniciais

Os *grids* computacionais disponibilizam um alto poder de processamento e armazenamento. Esse poder computacional geralmente está distribuído em vários recursos heterogêneos, não dedicados e pertencentes a diferentes domínios.

As estratégias de paralelização mais típicas enfatizam apenas o problema a ser paralelizado, sem preocupação alguma com as características do ambiente usado na execução. A estratégia mestre-trabalhador pode, por exemplo, para determinada aplicação, ter um excelente desempenho quando executada em um ambiente de *cluster* com poucas máquinas, e ter um desempenho menos satisfatório ao ser executada em um *grid* com um número maior de máquinas.

Essa diferença de desempenho pode ocorrer por vários motivos, um dos principais é o possível gargalo criado no processo mestre. Com um número maior de recursos, cresce a quantidade de processos trabalhadores e, conseqüentemente, o número de mensagens para o mestre também aumenta. Isso demonstra que as estratégias tradicionais não incorporam escalabilidade

como uma de suas características.

Para uma execução eficiente nos ambientes *grid*, faz-se necessário que os algoritmos paralelos sejam desenvolvidos através de estratégias de paralelização capazes de se adaptarem as características específicas do ambiente e a sua natureza em grande escala. Além da estratégia de paralelização adequada, o bom desempenho de uma aplicação em ambiente *grid* depende, de um mecanismo responsável por monitorar e gerenciar quaisquer alterações no ambiente, tais como mudanças na carga de alguma máquina ou na disponibilidade dos recursos.

Esse monitoramento pode ser feito diretamente pelo programador durante o desenvolvimento da aplicação ou por meio de *middleware*. No primeiro caso, as funções de gerência são parte integrante da aplicação e sempre que uma nova implementação for desenvolvida, novamente a gerência terá que ser tratada, elevando a dificuldade no desenvolvimento dessas aplicações. No segundo caso, como as funções de gerência estão em uma camada adicional, a cada nova aplicação, faz-se necessário apenas integrar as funções de gerenciamento com a aplicação.

A integração entre a estratégia proposta e o *middleware* desenvolvido é capaz de transformar as metaheurísticas paralelas em aplicações autonômicas, capazes de se adaptarem as variações do *grid*. As próximas seções descrevem a estratégia alternativa e o *middleware* propostos nesta tese.

6.2 Estratégia de Paralelização Hierárquica Distribuída

A estratégia hierárquica distribuída apresentada nesta seção surgiu a partir da observação feita com as implementações paralelas discutidas no Capítulo 5. Notou-se que a cooperação entre os processos de uma aplicação paralela leva a soluções melhores e em menores tempo de processamento. Dessa forma, a necessidade de desenvolver um padrão de implementação que pudesse implantar cooperação entre processos, sem desgaste no desempenho tornou-se evidente.

O objetivo da estratégia proposta é estruturar a aplicação de tal maneira que ela possa facilmente se adaptar ao ambiente *grid*, garantindo um bom nível de cooperação entre os processos da aplicação, independentemente da escala do ambiente, da heterogeneidade dos recursos e do número de processos necessários para concluir a execução.

Com esta estratégia, a aplicação a ser paralelizada é subdividida hierarquicamente em três níveis, como apresentado na Figura 6.1. Quanto maior for o número de recursos disponíveis no ambiente *grid*, maior será o número de

processos dos dois últimos níveis. No primeiro nível há uma única lista global, chamada lista de cooperação (LC), responsável por armazenar informações a serem trocadas entre os sítios do ambiente *grid*.

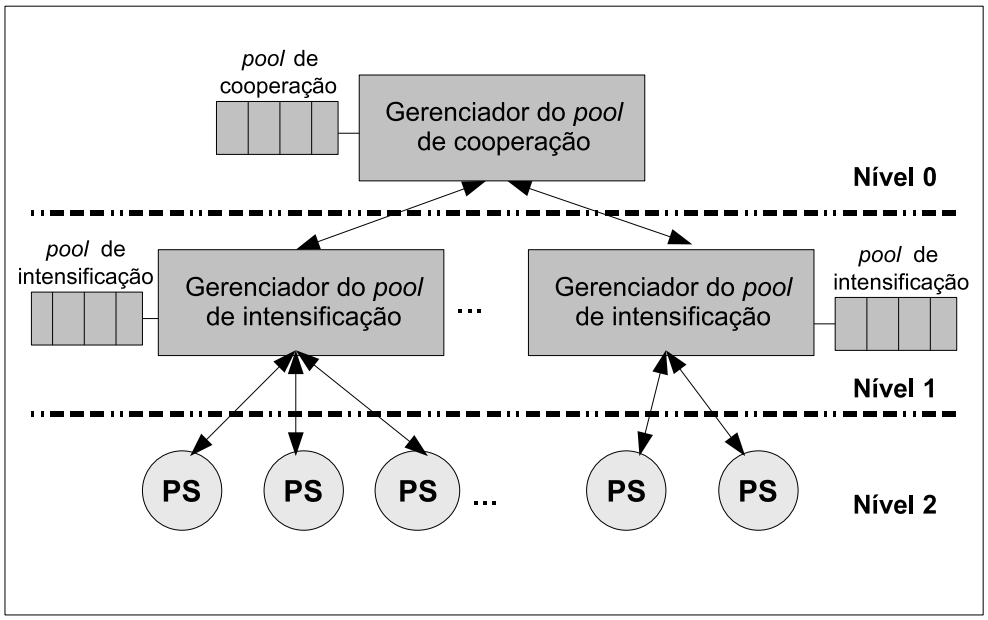

Figura 6.1: Estratégia de paralelização hierárquica distribuída.

No nível seguinte, há uma lista local em cada sítio do ambiente participante da execução, chamada lista de intensificação (LI). Esse tipo de lista é responsável por armazenar as melhores soluções encontradas pelos processos que estão executando em máquinas locais.

O último nível é formado pelos processos provedores de solução (PS) que executam efetivamente as heurísticas. Eles executam independentemente dos demais, aproveitando a estrutura hierárquica das listas criadas para garantir a cooperação com todos os processos provedores de solução, sem que isso acarrete uma sobrecarga ao sistema. A função específica de cada provedor dependerá do problema a ser resolvido pelos mesmos. As próximas seções descrevem as principais funções desempenhadas em cada nível da estratégia.

6.2.1 Provedor de Solução - PS

É neste nível que o usuário deve implementar as heurísticas que vão gerar as soluções do problema, de maneira que as mesmas possam aproveitar eficientemente a estrutura hierárquica de listas proposta para a troca de informação.

Os provedores de solução são independentes e assíncronos, mas compartilham informações através das listas de soluções de elite. Eles podem ser homogêneos ou heterogêneos: em uma mesma execução podem haver provedores

de solução executando heurísticas iguais ou diferentes. Nesse sentido, é possível expandir essa heterogeneidade para que se tenha também diferentes métodos, como, por exemplo, métodos exatos e heurísticos executando paralelamente na resolução do mesmo problema.

Além disso, os processos provedores de solução podem optar por iniciar suas execuções com soluções armazenadas na lista de intensificação associada ao sítio no qual ele está executando, ou iniciar a partir de alguma heurística construtiva. No primeiro caso, é necessário que o provedor mande uma mensagem para o processo que controla a lista de intensificação solicitando uma ou mais soluções de elite nela armazenada, que será escolhida, por exemplo, de maneira aleatória. Após receber a solução, o processo provedor de solução inicia sua execução objetivando melhorá-la. No segundo caso, o processo provedor constrói uma nova solução.

Em ambos os casos, o processo provedor envia a solução encontrada para ser armazenada na lista de intensificação local de soluções de elite. Isso significa que, uma das funções dos processos provedores é alimentar essa lista com as suas soluções.

6.2.2 Gerenciador da Lista de Intensificação - LI

O processo gerenciador da lista de intensificação (ou local) é responsável por três operações: inserir novas soluções em posições apropriadas da lista, escolher uma solução da lista (de acordo com a demanda dos processos provedores de soluções), e abastecer a lista de cooperação (ou global) com as melhores soluções existentes na sua lista.

A primeira operação é ativada todas as vezes que o gerenciador da lista de intensificação recebe uma solução a partir de um processo provedor. Nesse caso, a decisão de inserir ou não a nova solução deve ser tomada, a fim de que a diversidade seja mantida. Inicialmente, essa decisão é baseada na origem da solução recebida: se a solução foi gerada a partir de outra solução da lista, ou se a solução foi gerada pelo processo provedor de soluções a partir de alguma heurística construtiva. Se a solução recebida for melhor do que a solução de elite a partir da qual ela foi gerada, a nova solução substituirá a original. Por outro lado, se a nova solução tiver sido gerada por uma fase construtiva, ela poderá ser inserida diretamente em alguma posição disponível da lista.

Quando a lista está totalmente preenchida, a nova solução é inserida seguindo critérios definidos pelo desenvolvedor. Uma das opções é inserir a nova solução somente se ela for melhor do que a pior mantida na lista. Em ambos os casos, a nova solução é comparada com todas as demais já armazenadas, pois

só haverá inserção se não houver alguma outra solução com o mesmo custo. O critério usado para comparar as soluções deve ser definido pelo desenvolvedor, pois depende do problema.

A segunda operação ocorre todas as vezes que um processo provedor solicita uma solução de elite. Caso a lista de intensificação ainda esteja vazia ou não tenha solução suficiente, o processo gerenciador da lista informa ao processo solicitante que não há solução suficiente para ser enviada. Nesse caso, o processo provedor solicitante é obrigado a iniciar sua execução construindo uma nova solução. Quando a lista não está vazia, o gerenciador da lista escolhe uma solução de elite e a envia ao processo solicitante. Uma opção é que essa escolha seja feita aleatoriamente, através de uma distribuição uniforme.

A terceira operação ocorre todas as vezes em que uma lista de intensificação é completada. Imediatamente após o preenchimento, as melhores soluções da mesma (referenciadas como *grupo seleto*) são enviadas para serem armazenadas na lista de cooperação. Após enviar o grupo seleto de soluções, é importante que o processo gerenciador da lista de intensificação envie as atualizações do grupo seleto para o gerenciador da lista de cooperação. Para isso, todas as vezes que o gerenciador da lista de intensificação receber uma nova solução que substitua uma das soluções do grupo seleto enviada, ele deve enviar imediatamente a nova solução para atualização na lista de cooperação. Dessa forma, essa lista sempre manterá as melhores soluções encontradas em todos os sítios envolvidos na execução da aplicação.

As metaheurísticas tendem a um estado de estabilização, onde melhorias das soluções existentes ocorrem com pouca freqüência. O processo gerenciador da lista de intensificação é dito ter estabilizado quando um dado número max de mensagens com soluções consecutivas tiver sido recebido de seus provedores com soluções que não tenham sido aproveitadas para inclusão na lista.

Para escapar do estado de estabilização, o processo gerenciador da lista de cooperação deve executar um procedimento de *renovação*. Nesse caso, o gerenciador da lista de intensificação recebe uma quantidade de soluções de elite selecionadas aleatoriamente da lista de cooperação, para armazenar em sua lista. As demais posições da lista são esvaziadas, objetivando liberar espaço para que novas e diferentes soluções possam ser gravadas. Esse valor pode ser definido em tempo de execução pelo usuário através de parâmetros.

Com as novas soluções armazenadas na lista de intensificação, é possível que alguns provedores de soluções reiniciem suas execuções a partir de alguma nova solução dessa lista. Conseqüentemente, pode ocorrer que soluções geradas por processos que executaram em outros sítios sejam enviadas para o sítio que estabilizou, através da lista de cooperação. Assim, a estratégia proposta

permite a cooperação entre processos de sítios diferentes, o que explica o nome *lista de cooperação*, vista na próxima seção.

6.2.3 Gerenciador da Lista de Cooperação - LC

No topo da hierarquia, há o processo gerenciador da lista de cooperação (ou global), cuja principal função é garantir a troca de informações entre os processos provedores de solução que estão em diferentes sítios. Essa cooperação entre os processos é fundamental para melhorar o desempenho da aplicação.

Essa lista é criada inicialmente vazia. As regras para inserção de novas soluções são similares àquelas para a lista de intensificação. Novas soluções são inseridas em uma posição vazia. Quando a lista está preenchida, as novas soluções atualizam as piores. Dessa forma, a lista de cooperação é mantida exclusivamente pelas soluções enviadas por cada sítio, tornando-se um depósito das melhores soluções encontradas durante a execução da aplicação.

Uma das funções do processo que mantém a lista de cooperação é ativar a operação de renovação sempre que uma mensagem de estabilização for recebida de alguma lista de intensificação. Como já mencionado, a operação de renovação consiste em selecionar aleatoriamente um certo número de soluções e enviá-las para o gerenciador da lista de intensificação que tiver estabilizado. Todavia, quando a lista de cooperação está vazia, ou só tem as soluções enviadas pelo processo que estabilizou, o gerenciador da lista de cooperação manda uma mensagem de renovação total. O gerenciador estabilizado ao receber essa mensagem deve esvaziar completamente a sua lista e continuar sua execução com a lista vazia.

Com a operação de renovação, o gerenciador da lista de cooperação possibilita a troca de informação entre processos que executam em diferentes sítios do ambiente *grid*. Dessa forma, é possível realizar uma ampla troca de informação entre todos os processos da aplicação.

A estrutura distribuída e hierárquica de listas faz com que a maior parte da comunicação aconteça localmente, entre os processos que executam no mesmo sítio, sem eliminar a possibilidade de processos de um sítio terem acesso a informações geradas em outro, através da técnica de renovação. Em ambiente *grid*, isso significa que a maioria das mensagens são trocada em redes locais, ou seja, dentro de um mesmo sítio envolvendo os processos provedores e o seu gerenciador de lista. Com isso, a sobrecarga de comunicação não causa grande interferência no desempenho da aplicação executada. Conseqüentemente, essa estratégia pode ser usada para trabalhar com processos que implementem muitas ou poucas trocas de mensagens, independentemente do número de

processos criados.

Outra vantagem da estratégia proposta é a sua capacidade de tratar naturalmente a heterogeneidade do ambiente *grid*. Com a estrutura de listas não há necessidade de algum processo gerenciador da lista de intensificação ficar sincronizado com os processos provedores, esperando que um processo específico execute. O gerenciador da lista de intensificação pode ser programado para esperar que determinada ação ocorra na lista, independente do processo. Assim, se houver alguma máquina com capacidade de processamento menor que as demais, isto, em nada interferirá a execução da aplicação, pois as máquinas mais rápidas podem gerar as ações que o gerenciador aguarda. Um exemplo de ação esperada por um gerenciador de listas pode ser por exemplo, que um determinado número de soluções sejam armazenadas na lista.

A escalabilidade também é tratada pela estratégia proposta, pois quanto maior o número de máquinas disponíveis no ambiente, maior será o número de processos provedores. Para evitar um congestionamento nos processos gerenciadores das listas de intensificação é necessário apenas aumentar seu número, sem que nenhuma outra mudança na estrutura da aplicação seja necessária.

Contudo, nota-se que questões relacionadas diretamente com o ambiente devem ser tratadas para que a aplicação possa ser executada eficientemente. Nesse contexto, este trabalho de tese propõe o uso de um sistema de gerenciamento para tratar todas as características que independem da aplicação. O sistema de gerenciamento proposto foi implementado através de um *middleware*, cujas características específicas são apresentadas na Seção 6.4.

6.3 Políticas de Gerência da Estratégia

Ao implementar uma metaheurística paralela com a estratégia proposta na Seção 6.2, o desenvolvedor deve definir quatro políticas de gerência das listas: políticas de atualização, substituição, seleção e cooperação.

Essas políticas devem ser traçadas de acordo com as características do problema que estiver sendo paralelizado. Os detalhes específicos de cada uma dessas políticas são tratados a seguir.

Política de Atualização

A política de atualização define quando os processos provedores de solução devem mandar informações para serem armazenadas na lista de intensificação ao qual ele é vinculado. Dentre as opções, destacam-se:

– a cada nova solução gerada pelo provedor de solução;

– após um certo número de iterações realizadas, enviar a melhor;
– ao finalizar a execução, enviar a melhor solução encontrada;

Política de Substituição

Após estabelecer quando o processo provedor de solução enviará informações para a sua lista de intensificação, deve-se definir a política de substituição. Essa política só é usada quando a lista local está completamente preenchida. Enquanto houver espaço, a única regra para inserir uma nova solução é que não exista outra igual. Assim, quando o processo gerenciador da lista de intensificação receber uma nova solução, após a lista ter sido completamente preenchida, ele deverá decidir qual solução será substituída. Algumas possibilidades são:

– a pior solução;
– a solução mais antiga; e
– a solução mais parecida com a nova solução.

Política de Seleção

O próximo passo é o usuário definir a política de seleção. Essa tem a responsabilidade de indicar o critério usado pelo gerenciador da lista de intensificação para escolher uma solução e enviar para algum processo provedor de soluções. Algumas possibilidades são:

– selecionar aleatoriamente;
– escolher a solução mais antiga;
– escolher a melhor solução da lista; e
– criar uma lista restrita de candidatos e escolher a partir dela.

Política de Cooperação

Por último, devem ser definidas as regras de utilização dessas informações na lista. Isso é estabelecido pela política de cooperação. Essas regras permitem que o algoritmo se beneficie da cooperação entre os processos provedores de solução. Algumas opções podem ser:

– uma fração dos processos criados inicializa com soluções a partir da lista de intensificação, e a outra parte inicializa criando novas soluções;
– após um certo número de iterações, todos os processos realizam busca local com alguma solução a partir da lista; e

– escolhe aleatoriamente quais processos vão iniciar suas buscas com soluções a partir da lista de intensificação.

Para que a cooperação possa contribuir efetivamente na aceleração da convergência para boas soluções, a lista de cooperação deve manter as melhores soluções de cada lista de intensificação. Dessa forma, a Figura 6.2 ilustra o procedimento de preenchimento da lista de cooperação, em função do número de listas de intensificação. Nesse exemplo, a quantidade de soluções enviada foi igual a 1/3 do tamanho da lista de cooperação.

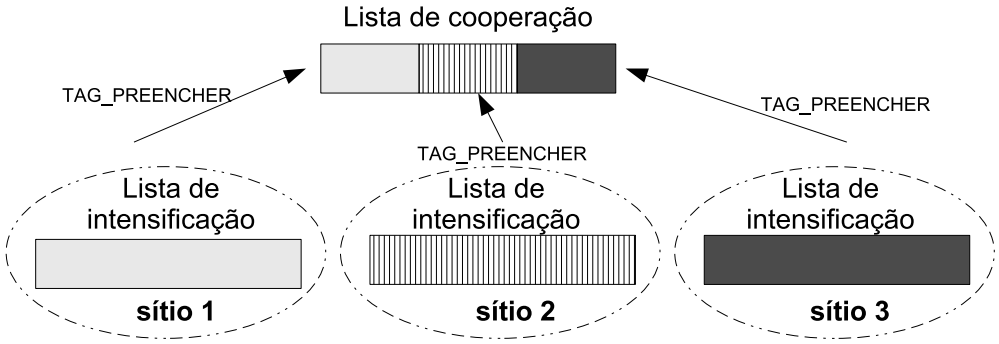

Figura 6.2: Exemplo de uma ação de preenchimento da lista de cooperação, com três listas de intensificação.

Após a lista de cooperação ser preenchida, uma nova solução só será inserida quando ela for melhor do que a pior solução armazenada na lista. Além disso, essa solução deve ser diferente de todas as demais.

Outro importante parâmetro que o usuário deve definir, em relação à lista de cooperação, é a fração de renovação que será usada em cada operação de renovação. Após armazenar as soluções recebidas a partir da lista de cooperação, o gerenciador que tiver estagnado deve esvaziar as demais posições na sua lista. A Figura 6.3-(a) apresenta uma operação de renovação em que a fração de renovação foi definida em 25% e a Figura 6.3-(b) apresenta uma operação com 50%.

Essa é uma decisão relevante porque se o usuário definir uma fração muito grande, por exemplo 80% das soluções, restará pouco espaço para que as listas de intensificação aproveitem suas soluções geradas localmente a partir da operação de renovação. Por outro lado, se a fração for pequena, por exemplo 25% como no exemplo da Figura 6.3-(a), haverá pouco aproveitamento das soluções geradas em sítios diferentes.

Conforme indicado nos dois exemplos da Figura 6.3, o gerenciador da lista de cooperação seleciona aleatoriamente as soluções que serão enviadas ao gerenciador estagnado. A escolha aleatória das soluções enviadas é adequada para proporcionar diversidade nas buscas realizadas. Dessa forma, é possível

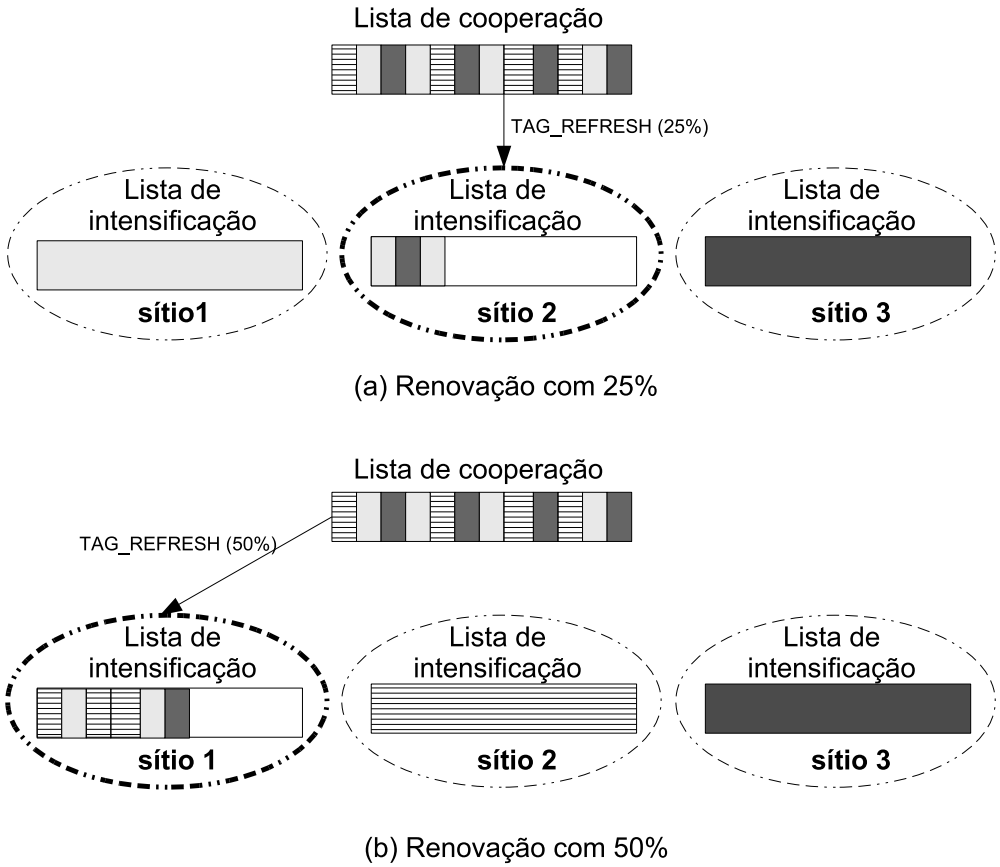

Figura 6.3: Exemplos de operações de renovação nas listas de intensificação.

que um processo executando em um determinado sítio compartilhe informações geradas em todos os demais pertencentes ao ambiente.

Tamanho das Listas

O tamanho de cada lista na hierarquia proposta é um fator importante. Esse tamanho pode ser definido diretamente pelo próprio desenvolvedor da aplicação paralela através de parâmetro, ou por um gerenciador do ambiente em que a aplicação será executada.

Quando é especificado pelo desenvolvedor tem a vantagem de ser definido de acordo com as características da aplicação, por exemplo aplicações de alta escala podem adotar listas grandes e aplicações mais rápidas podem assumir listas menores durante a execução. Por outro lado, quando é definido pelo gerenciador o tamanho de cada lista pode ser definido através do número de recursos disponíveis no ambiente de execução.

Experimentos computacionais realizados durante esta tese indicam que o tamanho das listas deve ser proporcional ao número de máquinas envolvidas na execução da aplicação. Além disso, as listas não devem ser grandes demais porque tendem a armazenar soluções não muito boas. Por outro lado, as listas

muito pequenas podem resultar em vários processos trabalhando nas mesmas soluções, pois as opções de soluções a serem usadas nas operações de renovação tornam-se reduzidas.

Como dito anteriormente, no desenvolvimento de um programa paralelo é fundamental que o ambiente usado na execução seja conhecido. Por serem compostos por recursos heterogêneos e pela sua natureza dinâmica, onde os recursos disponíveis podem ser compartilhados e estarem geograficamente distantes, os *grids* demandam uma gerência dos seus recursos.

Para que possa haver um bom aproveitamento do potencial computacional dos ambientes de *grid*, é importante que seu comportamento seja monitorado e a execução da aplicação gerenciada. A Seção 6.4 descreve detalhadamente o sistema de gerenciamento proposto.

6.4 Sistema de Gerenciamento - metaEasyGrid

O sistema de gerenciamento é responsável por tratar todas as questões específicas do ambiente. Dentre suas principais funções destacam-se o balanceamento de carga, a tolerância a falhas, a descoberta de recursos, a criação de tarefas e o monitoramento do ambiente. Para garantir a viabilidade no uso do ambiente é importante que todas as funções de gerência sejam realizadas de maneira transparente à aplicação.

Para garantir que o usuário e o desenvolvedor possam ficar isentos dos detalhes específicos à gerência, o sistema de gerenciamento foi implementado na forma de um *middleware*. Nesta tese adotou-se o *middleware* SGA EasyGrid [115] (apresentado na Seção 4.3) para gerenciar a execução das aplicações no ambiente *grid*.

O SGA EasyGrid é um *middleware* capaz de transformar automaticamente aplicações paralelas em implementações autonômicas e *system-aware*. Aplicações desse tipo são adaptativas, robustas à falhas de recursos e capazes de reagirem às mudanças do sistema que podem ocorrer em ambientes dinâmicos.

Além disso, uma característica chave desse *middleware* é o seu modelo de execução [33]. As aplicações devem ser escritas de tal maneira que o paralelismo disponível seja maximizado, independentemente do número de processadores disponíveis. Embora o modelo tradicional de "um processo por processador" seja eficiente para ambientes dedicados e homogêneos (como os *clusters*), uma implementação que usa um maior número de processos menores pode obter melhor desempenho no ambiente *grid*, apesar das sobrecargas causadas pela criação de processos [33]. A solução encontrada para minimizar os efeitos da sobrecarga gerada com a criação de vários processos foi aumentar a execução

concorrente nos processadores. Isso é feito através do aumento no número de processos leves disparados no mesmo processador. Assim a carga gerada pela criação de um processo torna-se transparente porque acontecerá junto com a execução de um outo processo. Quando um novo processo estiver sendo criado, o processador não estará ocioso esperando pela criação.

De acordo com a estratégia de paralelização aplicada às metaheurísticas e com o modelo de execução adotado pelo SGA EasyGrid, em que o número de processos a ser criado deve ser proporcional ao grau de paralelismo da aplicação e ao trabalho a ser feito, não seria possível especificar exatamente o número total de processos a ser criado durante a execução de uma metaheurística paralela.

Esse total de processos depende da combinação de vários fatores, tais como número usado como semente e os números aleatórios gerados na aplicação. Dessa forma, definir antes da execução de uma metaheurística o número total de processos que deveriam ser criados até que o trabalho total fosse concluído não é um trabalho viável.

Para que o SGA EasyGrid fosse capaz de gerenciar a execução de metaheurísticas paralelas em ambientes *grid* sem conhecer antecipadamente o número de tarefas a serem criadas, fez-se necessário realizar algumas alterações na sua versão inicial. O mesmo estava preparado só para gerenciar aplicações BoTs, nas quais o número total de tarefas que a aplicação precisa criar é conhecido antecipadamente.

Na nova versão do SGA EasyGrid, chamada SGA metaEasyGrid, a quantidade de processos criada ao longo da execução é indeterminada a priori, mas limitada por condições de parada especificadas na própria metaheurística. Nesse caso, como o *middleware* não tem conhecimento do número total de processos que serão criados até a aplicação ser concluída, ele cria dinamicamente os processos em blocos. Optou-se pela criação em bloco para evitar que a todo instante um novo processo tivesse que ser criado, sobrecarregando os gerenciadores só com a criação de processos. O tamanho de cada bloco pode variar de acordo com o número de processadores disponíveis no momento da criação e da carga de cada processador.

O SGA metaEasyGrid objetiva transformar metaheurísticas em aplicações capazes de se auto-adaptarem às mudanças dinâmicas que ocorrem em um ambiente *grid*. O projeto interno de cada processo de gerenciamento do SGA metaEasyGrid, foi mantido igual ao da versão original do SGA EasyGrid [32]. Ambos implementam uma hierarquia de camadas, onde cada serviço é realizado por um sub-sistema específico. Os mecanismos de escalonamento dinâmico [115] e de tolerância a falhas [139] são similares ao do

SGA EasyGrid, mas algumas alterações se fizeram necessárias para que esses pudessem se adaptar ao SGA metaEasyGrid. As próximas seções destacam as mudanças realizadas em cada sub-sistema específico do SGA EasyGrid.

6.4.1 Criação dos Processos

Tanto no SGA EasyGrid quanto no SGA metaEasyGrid, os processos são criados individualmente, ou seja, cada processo é criado a partir de uma chamada à função $MPI\_Comm\_spawn()$. Cada processo disparado recebe o mesmo identificador (*rank* 0) e um comunicador diferente dos demais, o que não permite a comunicação direta entre eles. Assim, toda comunicação entre processos da aplicação é realizada através dos processos gerenciadores de acordo com as suas hierarquias.

Desde o projeto do SGA EasyGrid, optou-se por criar cada processo individualmente com seu próprio comunicador porque, em caso de falha, somente o processo que falhou é eliminado, podendo ser recriado em outra máquina [139]. Assim, não há propagação da falha, pois ela é facilmente isolada. Além disso, a criação individual de cada processo viabiliza a comunicação direta entre qualquer par de tarefas. Contudo, é necessário que o processo transmissor conheça o comunicador do processo receptor. Como conseqüência, os processos gerenciadores do ambiente precisam ter conhecimento de todos os processos que estão sob sua gerência.

Nesse contexto, para cada processo GS disparado, o processo GG realiza uma chamada à função $MPI\_Comm\_spawn()$, o que faz com que cada processo criado tenha o identificador zero (*rank* 0) e um comunicador único. O fato de existirem comunicadores diferentes implica que a comunicação entre os GSs deve ser feita com a intervenção do GG. Assim, cada tarefa da aplicação disparada individualmente pelo processo GM é considerada como um processo independente e incapaz de se comunicar diretamente com os demais processos. A única comunicação possível do processo da aplicação é com o processo GM local, que é seu processo pai por ter sido quem o criou.

Quando há necessidade de comunicação entre processos da aplicação que estão em máquinas distintas, a participação dos gerenciadores de nível mais alto é fundamental.

A Figura 6.4 apresenta um exemplo com um GG, dois GSs, quatro GMs e sete tarefas da aplicação. Considera-se que a tarefa $T0$ envia uma mensagem para $T1$. Essa mensagem é redirecionada para o GM local, que ao recebê-la verifica se a tarefa destino $T1$ pertence à lista de tarefas escalonadas na máquina local. Ao perceber que a tarefa destino não foi atribuída à máquina

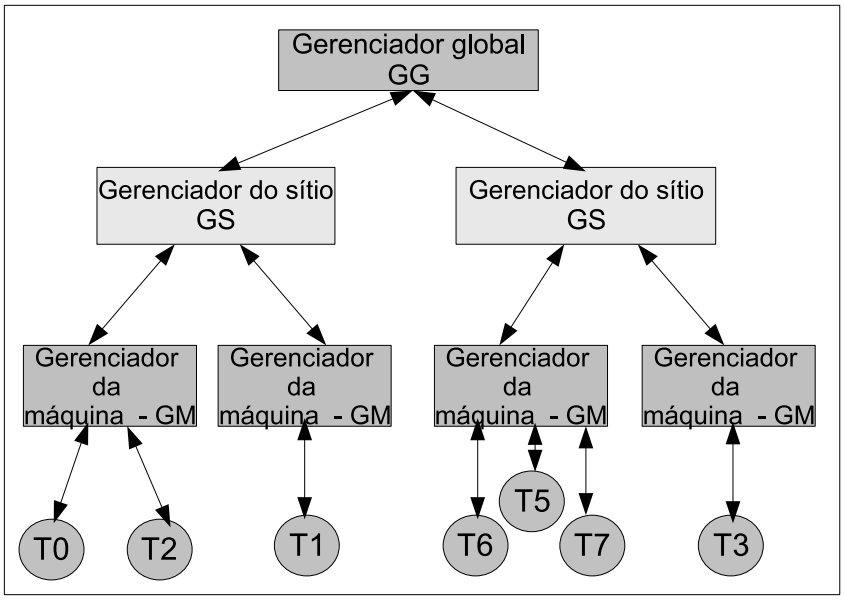

Figura 6.4: Exemplo de uma aplicação com sete tarefas, gerenciada pelo SGA metaEasyGrid.

local, o GM imediatamente redireciona a mensagem para o GS local, que é responsável por identificar e repassar a mensagem para o GM da máquina destino. Quando o GM destino recebe a mensagem, ele tem a responsabilidade de enviá-la para a tarefa destino $T1$.

Quando a troca de mensagem acontecer entre sítios, a mensagem é redirecionada do seu GM para o GS local, que ao perceber que a tarefa destino não foi atribuída a máquina alguma do seu sítio, imediatamente encaminha essa mensagem para o GG, que é o único processo da hierarquia de gerenciadores capaz de fazer comunicação entre sítios. Dessa forma, o GG verifica em qual GS está o processo destino e repassa a mensagem, que em seguida será reencaminhada para o GM destino.

Nessa abordagem, apenas o processo GG precisa ter conhecimento de toda a lista de tarefas da aplicação com suas respectivas alocações. Os demais processos da hierarquia não precisam conhecer todas as localizações das tarefas. O GM precisa apenas da lista de tarefas atribuídas à máquina local, uma vez que a comunicação envolvendo máquinas distintas do mesmo sítio é intermediada pelo GS. Assim, o GS precisa apenas da lista de todas as tarefas da aplicação atribuídas a cada máquina pertencente ao sítio gerenciado por ele [150].

Na versão SGA EasyGrid, o GG desempenha um papel ativo na criação das tarefas. Após criar as estruturas de dados relativas ao gerenciamento do ambiente, ele envia para os GSs todo o conjunto de tarefas que cada sítio executará.

O GG tem um papel passivo no SGA metaEasyGrid, pois só criará tarefas após algum GS solicitá-las. Nesse caso, logo depois que o GS é criado, ele solicita ao GG um bloco de tarefas. O tamanho desse bloco é proporcional à capacidade de processamento do sítio. Ao receber a mensagem para criar o bloco de tarefas, o GG atualiza dinamicamente suas estruturas de dados (que dependem do número de processos criados), e encaminha ao GS a sua nova lista de tarefas.

Após essa criação inicial, todas as vezes que o GS receber a finalização de uma tarefa, ele verificará a carga (o número de processo prontos para serem executados) do seu sítio. Assim, quando a carga for baixa ele solicitará ao GG a criação de um novo bloco, antes que os recursos do seu sítio fiquem ociosos. Na implementação realizada do SGA metaEasyGrid, considerou-se carga baixa de um sítio quando o número de tarefas prontas em cada um atinge o valor:

$$lim\_inf = 2. \left(\frac{blocoTarefas}{numTarefaProcessador} \right)$$

sendo, $blocoTarefas$ o tamanho do bloco de tarefas criado em cada sítio e $numTarefaProcessador$ a quantidade de tarefas disparadas em cada processador de uma só vez pelo *middleware* através do GM.

A principal vantagem de criar novas tarefas sempre que a carga de um sítio atingir um limite mínimo é evitar que máquinas fiquem ociosas, esperando que o *middleware* crie novas tarefas. Além disso, o tempo de criação dos processos e de atualização das estruturas de dados fica sobreposto ao processamento dos mesmos.

6.4.2 Balanceamento de Carga

Como visto na Seção 4.3.5, o SGA EasyGrid implementa um escalonamento de processos híbrido. Inicialmente, o escalonador estático distribui as tarefas entre as máquinas dos sítios. Durante a execução da aplicação, o escalonamento dinâmico pode ser ativado para reajustar a carga no ambiente.

O escalonador estático não é usado no caso do SGA metaEasyGrid, pois nenhuma tarefa da aplicação é criada estaticamente. Assim sendo, quando as tarefas da aplicação forem criadas é necessário ativar o escalonador dinâmico para decidir em qual máquina cada tarefa deve ser inicializada.

Outra diferença em relação ao balanceamento de carga do SGA metaEasyGrid é que o escalonamento dinâmico é realizado só em dois níveis. Como os blocos de processos são criados sob demanda para cada sítio específico, quando a carga de algum deles for inferior a lim_inf, novas tarefas

serão criadas sem que haja necessidade de realizar re-escalonamento de tarefas entre os mesmos. A vantagem é que não há uma possível sobrecarga causada pelo re-escalonamento das tarefas prontas entre os sítios.

Contudo, todas as vezes que o sítio receber novos processos, o escalonador dinâmico do mesmo é ativado imediatamente a fim de distribuir as novas tarefas entre suas máquinas. Assim, cada GS monitora o estado do sítio, mantendo sua carga balanceada de acordo com seu poder computacional.

O último nível do mecanismo de balanceamento dinâmico também é usado no SGA metaEasyGrid. Todas as vezes que um GM receber tarefas alocadas à sua máquina, ele acionará o escalonador dinâmico da máquina para determinar a ordem em que elas serão executadas. Com esse escalonamento dinâmico pró-ativo, a probabilidade de máquinas ficarem ociosas é reduzida.

O escalonador dinâmico da máquina também é responsável por controlar o número de tarefas que simultaneamente pode ser executado em um mesmo recurso. Isso é controlado pelo *middleware* com o objetivo de otimizar o desempenho da aplicação.

6.4.3 Tolerância a Falhas

Em relação à detecção de falhas, nada foi alterado no SGA metaEasyGrid. As mudanças realizadas estão relacionadas à recuperação dos processos. No SGA EasyGrid, quando um processo da aplicação falha, o mesmo é recriado e, imediatamente, todas as mensagens que haviam sido encaminhadas a ele são devidamente re-encaminhadas. Contudo, para cada processo é necessário manter uma lista com todas as mensagens recebidas, o que pode provocar um grande consumo de memória se os processos realizarem muitas trocas de mensagens [139].

No caso de metaheurísticas, os processos caracterizam-se por terem comportamentos diferentes em cada execução. Dessa forma, as mensagens trocadas por um processo em uma execução podem não mais serem geradas na próxima. Nesse caso, a única solução para manter uma re-execução igual aquela que falhou seria a realização de *checkpoint* dos processos.

Como nas metaheurísticas o tempo de processamento de uma tarefa pode variar consideravelmente em função dos parâmetros usados, é possível que após reiniciar o processo que falhou esse esteja tão próximo do fim da sua execução que o tempo necessário para reiniciá-lo não compense o custo envolvido na realização do *checkpoint*. Além disso, com o modelo de execução adotado pelo SGA metaEasyGrid, os processos tendem a ser menores.

Dessa forma, a opção por reiniciar os processos ao invés de fazer *check-*

point foi tomada devido às características das metaheurísticas e do modelo de execução usado não compensarem implementá-lo.

No SGA metaEasyGrid sempre que for detectada a falha de algum processo, o mesmo é recriado imediatamente sem a necessidade de redirecionamento de mensagens. O mecanismo de tolerância a falhas tentará criá-lo sempre que possível na mesma máquina.

Com os processos de gerência do *middleware*, o único processo que não pode falhar é o GG. Por causa disso, tanto o SGA EasyGrid quanto o metaEasyGrid consideram que a máquina do GG deve estar preparada para tratar eventuais falhas. No caso de algum GS falhar, ao ser recriado ele verificará sua carga de tarefas prontas, se notar que estava vazia, ele imediatamente solicitará ao GG um novo bloco de tarefas. Por último, se algum GM falha, o GS recria esse processo e, em seguida, re-escalona tarefas de outro GM do mesmo sítio para a máquina que falhou [139].

6.4.4 Finalização dos Processos

Outra mudança implementada no SGA metaEasyGrid está relacionada à finalização dos processos gerenciadores. No SGA EasyGrid, o GG ao perceber que todas as tarefas foram executadas, manda uma mensagem de finalização para cada GS. Ao receber mensagem de fim a partir do GG, cada GS encaminha para os seus GMs associados. Os processos GMs finalizam suas execuções e, logo em seguida, os GSs também finalizam suas execuções ao notarem que não há GM algum associado ao seu sítio. Por último, o GG percebe que todos os GSs encerraram suas execuções, e finaliza [150].

No SGA metaEasyGrid o processo de finalização teve que ser alterado porque o GG não tem conhecimento do número total de tarefas que devem ser executadas. Além disso, sempre que a lista de tarefas prontas estiver chegando ao fim em cada sítio, os GSs solicitam a criação de novas. Dessa forma, o GG não pode esperar que todas as tarefas sejam executadas para finalizar. Como solução para isso, o GG só encerrará a sua execução quando receber uma mensagem do módulo de monitoramento, informando que o processo gerenciador da lista de cooperação finalizou. Nesse momento, o GG envia uma mensagem de finalização para todos os GSs. Em seguida, cada GS libera as filas de tarefas prontas (ainda não criadas) e pendentes.

Quando os GSs recebem essa mensagem de fim, eles imediatamente a propagam para todos os seus GMs locais. Ao ser notificado com a mensagem de parada, cada GM libera suas filas de processos prontos e pendentes e encerra também a sua execução. Dessa forma, a finalização de todos os processos da

aplicação que não foram executados torna-se completamente transparente ao usuário.

6.5 Integração da Estratégia de Paralelização com o SGA metaEasyGrid

Para que o usuário possa se beneficiar da gerência realizada pelo SGA metaEasyGrid é necessário apenas que a aplicação paralela seja compilada com o *middleware*. Assim, todas as funcionalidades do SGA metaEasyGrid são embutidas automaticamente na aplicação.

Além dos benefícios obtidos diretamente com o gerenciamento do ambiente, a aplicação paralela torna-se muito mais robusta porque adquire portabilidade, escalabilidade e autonomia.

A portabilidade torna-se uma realidade das aplicações paralelas, pois com o *middleware* embutido na própria aplicação ela poderá ser executada em qualquer ambiente *grid* que ofereça os serviços de gerenciamento básico como os realizados pelo Globus [65] e pela biblioteca de passagem de mensagens MPI [142].

Em relação à escalabilidade, a heurística paralela torna-se mais robusta, porque com o SGA metaEasyGrid sendo responsável pela criação das tarefas, não existe limite no número de tarefas que a aplicação pode criar. Como a criação de tarefas acontece sob demanda, de acordo com a disponibilidade dos recursos, o usuário não precisa se preocupar se há poder computacional ou memória suficientes para criar tantas tarefas. Em testes realizados com o *middleware* EasyGrid já foram executadas aplicações com até 100.000 tarefas [115]. Com o SGA metaEasyGrid foi executada uma aplicação durante 20 horas de processamento e tarefas que tinham tempo médio de processamento igual a 10 segundos, em 24 máquinas. Nesse caso, foram executadas mais de 180.000 tarefas.

Por último, a aplicação torna-se autonômica, porque se falhas ocorrerem ela é capaz de re-iniciar o processo que falhou sem que a aplicação inteira venha a falhar como acontece com aplicações MPI em geral. No caso da carga no ambiente mudar, as tarefas são re-escalonadas sem o auxílio de um escalonador externo, pois a aplicação usa a estrutura de balanceamento de carga embutida na própria aplicação pelo *middleware* [116].

Outra vantagem obtida com a integração da estratégia proposta e do SGA metaEasyGrid é a transparência alcançada com relação à gerência do ambiente. Quando o desenvolvedor desejar implementar outra aplicação para executar em *grid*, ele só precisa definir as políticas de execução da estratégia, definidas na Seção 6.3, e implementar o provedor de solução. Nenhuma função

de gerência do ambiente será alterada.

A integração entre a estratégia de paralelização e o SGA metaEasyGrid foi possível e natural porque ambos adotam uma arquitetura distribuída hierárquica das suas funcionalidades. Com isso, a integração ocorre sem acarretar sobrecarga na execução da aplicação paralela. Dessa forma, é possível garantir uma gerência eficiente, independentemente da escala do ambiente *grid* usado na execução da aplicação [115].

A Figura 6.5 mostra a integração entre as camadas do *milldeware* e a aplicação implementada através da estratégia proposta. Como pode ser notado, cada processo do SGA metaEasyGrid gerencia diretamente um nível da estratégia. Assim, todas as vezes que houver um aumento no tamanho da aplicação ou do ambiente, haverá também uma aumento no número de processos da aplicação e do *middleware*.

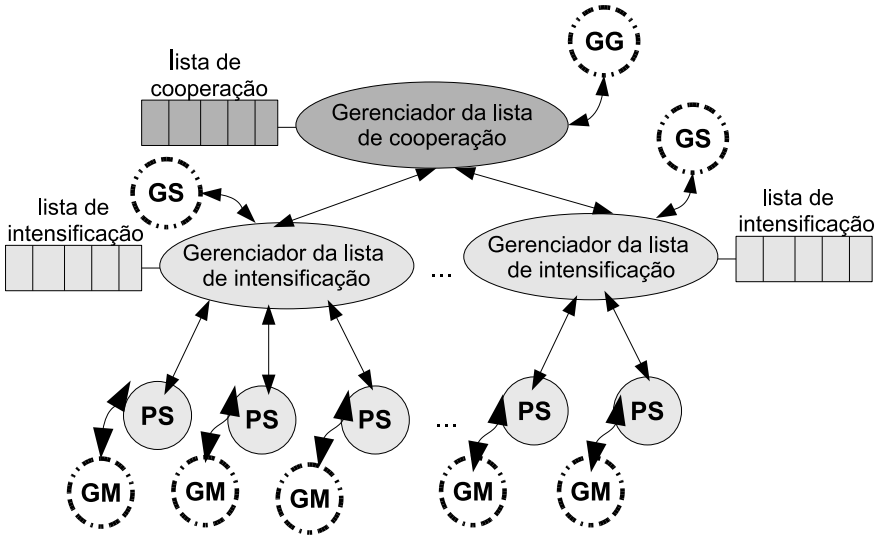

Figura 6.5: Integração da nova estratégia de paralelização com o SGA metaEasyGrid.

A fim de analisar os desempenho das heurísticas paralelas geradas pela combinação da nova estratégia proposta e ao SGA metaEasyGrid, foram implementadas, testadas e analisadas duas heurísticas paralelas. As Seções 6.5.1 e 6.5.2 destacam os detalhes de implementação dessas aplicações.

6.5.1 Implementação da Aplicação mTTP

A paralelização da heurística `GRILS-mTTP` através da estratégia proposta baseou-se no programa seqüencial desenvolvido em [147]. Para essa paralelização, chamada AUDImTTP (*Autonomic and Distributed* mTTP), foi necessário definir o gerenciador da lista de cooperação, o gerenciador da lista de intensificação e, por último, os provedores de solução.

Gerenciador da Lista de Cooperação

O Algoritmo 5 apresenta o pseudo-código do programa que gerencia a lista de cooperação da implementação paralela AUDImTTP. Inicialmente, na linha 4, a variável $solucaoMelhor$ é inicializada com valor infinito. Na linha 5, é criada a lista de cooperação, com tamanho proporcional ao número de máquinas disponíveis no *grid*. Em seguida, na linha 6 é lido o arquivo com os dados da instância do problema. O ciclo mais externo nesse algoritmo, formado pelas linhas de 7 até 42, executa até que um determinado critério de parada seja atingido. Nesse laço, a cada mensagem recebida proveniente de um dos gerenciadores das listas de intensificação, uma determinada ação é executada de acordo com o rótulo da mensagem. Caso a mensagem tenha o rótulo igual a $TAG\_ATUALIZA\_SOLUCAO$, o gerenciador receberá a solução na linha 11 e, caso não haja outra solução igual na lista, a mesma é inserida na primeira posição vazia. Se a lista já estiver lotada, a nova solução substituirá a pior. Essas regras são implementadas pela função `AtualizaListaGlobal()`, na linha 12. Se a condição determinada pela função `Substituicao()`, na linha 13 for satisfeita, a melhor solução conhecida no algoritmo é substituída na linha 14

Na linha 17 o rótulo $TAG\_PREENCHER\_LISTA$ da mensagem recebida significa que há novas soluções. Nesse caso, o gerenciador receberá na linha 18 um bloco de soluções. O bloco enviado contém um número de soluções que é uma fração do tamanho da lista de intensificação. O objetivo é fazer com que todos as listas de intensificação possam ter suas melhores soluções gravadas na lista de cooperação. Essa atualização da lista é feita pelo laço formado pelas linhas de 20 até 26. A decisão de armazenar (ou não) a solução recebida na lista de cooperação é tomada, na linha 23, pela mesma função `AtualizaListaGlobal()` da linha 12. Na linha 23, cada solução é avaliada se será aceita como a melhor solução conhecida pelo algoritmo. Em caso afirmativo, a melhor solução é substituída na linha 24.

Na linha 28, a mensagem recebida tem rótulo $TAG\_ESTAGNACAO$. O gerenciador verifica, na linha 30, se foi atingido o limite máximo de estagnação, definido em tempo de compilação. Em caso afirmativo, na linha 31 ele verifica se o critério de parada também foi atingido e, neste caso, na linha 32 ele armazena a melhor solução armazenada na lista em um arquivo de saída e, na linha 33 pára imediatamente a execução da aplicação. Caso contrário, o gerenciador da lista de cooperação inicializa o número de estagnação para que a execução possa continuar até atingir o critério de parada, linha 35. No caso do número máximo de estagnação não ter sido alcançado, o gerenciador realiza na linha 38 a operação de renovação, selecionando uma parte das suas soluções e as enviando para o gerenciador estagnado.

```
1  Algoritmo: global_AUDImTTP;
2  Entrada: tamanhoLista, instancia;
3  Saída: melhorSolucao;
4  solucaoMelhor ← INFINITO;
5  InicializaListaGlobal(tamanhoLista);
6  LerArquivoEntrada(instancia);
7  enquanto CriterioParada faça
8      tagMsg ← VerificarChegadaMensagem();
9      selecione tagMsg faça
10         caso TAG_ATUALIZA_SOLUCAO
11             solucao ← ReceberMsg();
12             AtualizaListaGlobal(solucao);
13             se Substituicao(solucao, solucaoMelhor) então
14                 solucaoMelhor ← solucao;
15             fim
16         fim
17         caso TAG_PREENCHER_LISTA
18             buffer ← ReceberMsg();
19             i = 0;
20             enquanto i < numSolucao faça
21                 i++;
22                 AtualizaListaGlobal(buffer[i]);
23                 se Substituicao(buffer[i], solucaoMelhor)
                   então
24                     solucaoMelhor ← buffer[i];
25                 fim
26             fim
27         fim
28         caso TAG_ESTAGNACAO
29             estagnacao++;
30             se estagnacao == MAX_ESTAGNACAO então
31                 se CriterioParada então
32                     SalvarMelhorSolucao();
33                     PararExecucao();
34                 fim
35                 estagnacao ← 0;
36             fim
37             senão
38                 Renovacao();
39             fim
40         fim
41     fim
42 fim
```

Algoritmo 5: Pseudo-código do gerenciador da lista de cooperação (ou global) para o AUDImTTP.

Gerenciador da Lista de Intensificação

O Algoritmo 6 detalha as operações realizadas pelo gerenciador da lista de intensificação. Inicialmente, na linha 4, é criada uma lista de intensificação em cada sítio. Na linha 5, os dados da instância são lidos. Em seguida, o gerenciador da lista entra em um laço, que inclui as linhas de 6 até 32, no qual fica esperando receber mensagens para tratá-las de acordo com seu rótulo. Essas mensagens podem ser recebidas tanto dos processos provedores de solução quanto do gerenciador da lista de cooperação.

A mensagem com rótulo TAG_ACABOU na linha 9, é enviada pelos processos provedores todas as vezes que finalizam sua execução. O gerenciador da lista de intensificação, ao receber essa mensagem na linha 10, atualiza a lista seguindo uma política de substituição implementada pela função `AtualizaListaLocal()` na linha 11. Essa função trata duas possibilidades: na primeira, se o processo provedor que enviou a solução tiver iniciado sua execução a partir da fase de construção GRASP, então essa solução substituirá a pior solução da lista, caso não haja outra solução igual. Na outra possibilidade, se o processo provedor tiver iniciado da fase de busca ILS, a solução recebida deverá ser armazenada exatamente na mesma posição da solução enviada ao processo provedor. O objetivo é diminuir a probabilidade de ter várias soluções parecidas na lista, geradas a partir da mesma solução.

Pode acontecer que a solução recebida não seja inserida na lista de intensificação, por exemplo por já ter uma solução igual ou por essa solução não ser melhor que a pior. Quando um certo número de soluções consecutivas são recebidas sem que alguma delas possa ser inserida na lista, é dito que o gerenciador da lista de intensificação estabilizou, linha 12. Nesse caso, é necessário que o gerenciador da lista de intensificação avise imediatamente ao gerenciador da lista de cooperação que ele está estabilizado. Isso é feito na linha 13 através da função `AvisaListaGlobalqueLocalEstagnou()`, que manda uma mensagem identificada como $TAG\_ESTAGNACAO$ para o gerenciador da lista de cooperação. Em seguida, na linha 14, o gerenciador da lista de intensificação aguarda pelo retorno da mensagem enviada.

Se o gerenciador da lista de cooperação tiver soluções suficientes para executar a operação de renovação, o mesmo enviará parte delas para o gerenciador estabilizado, com a mensagem $TAG\_REFRESH$, na linha 16. Caso contrário, ele mandará uma mensagem com TAG_VAZIO, linha 21. No primeiro caso, o gerenciador de intensificação receberá as soluções enviadas por meio da função `ReceberBlocoSolucao()`, linha 17. Em seguida, na linha 18 ele armazena as soluções recebidas na sua lista com a função `AtualizaListaLocal()`. E, na linha 19 as demais posições que não foram atualizadas são destruídas.

```
1  Algoritmo: local_AUDImTTP;
2  Entrada: tamanhoLista, instancia, totalSites;
3  Saída: ;
4  InicializaListaLocal(tamanhoLista);
5  LerArquivoEntrada(instancia);
6  enquanto CriterioParada faça
7      tagMsg ← VerificarChegadaMensagem();
8      selecione tagMsg faça
9          caso TAG_ACABOU
10             solucao ← ReceberMsg();
11             AtualizaListaLocal(solucao, estagnacao);
12             se estagnacao==MAX_ESTAGNACAO então
13                 AvisaListaGlobalqueLocalEstagnou();
14                 status=AguardaMsgRetorno();
15                 selecione status faça
16                     caso TAG_REFRESH
17                         ReceberBlocoSolucao();
18                         AtualizaListaLocal();
19                         ZerarParteListaLocal();
20                     fim
21                     caso TAG_VAZIO
22                         ZerarTodaListaLocal();
23                     fim
24                 fim
25             fim
26         fim
27         caso TAG_PED_SOLUCAO
28             EscolheSolucaoListaLocal();
29             EnviaSolucao();
30         fim
31     fim
32 fim
```

Algoritmo 6: Pseudo-código do gerenciador da lista de intensificação (ou local) do AUDImTTP.

No caso da mensagem recebida ter o rótulo TAG_VAZIO, o gerenciador estabilizado esvaziará toda a sua lista na linha 22 com a função `ZerarTodaListaLocal()`, possibilitando que futuras soluções geradas a partir de seus provedores de solução sejam armazenadas localmente.

Outra mensagem freqüentemente recebida pelos gerenciadores de intensificação também é gerada a partir dos provedores de solução. Essa mensagem é identificada pelo rótulo $TAG\_PED\_SOLUCAO$, na linha 27, e é enviada sempre que um processo provedor de solução for iniciar sua execução a partir da fase ILS. Nesse caso, o provedor de solução pede uma solução de elite a partir da sua lista de intensificação. Essa solução é selecionada através da função `EscolheSolucaoListaLocal()` ativada na linha 28, e enviada ao processo solicitador por meio da função `EnviaSolucao()` na linha 29. Este processo gerenciador executará até que seja atingido algum critério de parada. Isso ocorrerá quando o gerenciador da lista de cooperação tiver finalizado sua execução.

Provedores de Solução

Para finalizar a implementação paralela, é necessário desenvolvedor os processos provedores de soluções, que efetivamente irão executar a heurística. O Algoritmo 7 foi baseado na heurística híbrida `GRILS-mTTP` proposta por [147] e apresentada na Seção 2.3.2.

Cada provedor de solução inicializa na linha 4 gerando sementes individuais e únicas, baseadas no *rank* de cada processo. Na linha 5 é feito a leitura dos dados de entrada da instância. Em seguida, na linha 6, é feita uma escolha aleatória para definir se cada provedor de solução iniciará da fase de construção GRASP ou da fase de busca local ILS.

Essa seleção aleatória foi implementada para possibilitar maior cooperação entre os processos pois, assim, uma parte dos processos provedores inicia a partir da fase de construção GRASP para construir novas soluções, e outros iniciam sua execução a partir de soluções de elite, geradas por outros processos provedores e mantidas pelo gerenciador da lista de intensificação. Essa seleção é baseada em uma probabilidade $1 - Q$ para que os processos iniciem a partir da fase de construção GRASP e Q para que iniciem a partir da fase de busca ILS.

No segundo caso, os processos enviam na linha 8 uma mensagem identificada com rótulo $TAG\_PED\_SOLUCAO$ para o gerenciador da lista de intensificação. A resposta, recebida na linha 9, pode conter uma solução vazia, como indicado pela linha 10, e nesse caso a heurística terá que iniciar a partir da fase GRASP. Ou pode conter uma solução de elite, com a qual será atualizada todas as variáveis para a heurística começar na fase de busca ILS.

```
1  Algoritmo: provedor_AUDImTTP;
2  Entrada: instancia, semente, idSite;
3  Saída: solucao;
4  semente = semente + PegarMyRank();
5  LerArquivoEntrada(instancia);
6  FASE ← EscolheAleatoriamenteFase(Q);
7  se FASE == ILS então
8  |  EnviaMsgPedindoSolucao();
9  |  S = RecebeSolucaoListaLocal();
10 |  se S == NULL então
11 |  |  FASE ← GRASP;
12 |  fim
13 |  senão
14 |  |  AtualizaVariaveisComSolNova();
15 |  fim
16 fim
17 se FASE == GRASP então
18 |  S ← ConstrucaoGulosaAleatoria();
19 |  S, S ← BuscaLocal(S);
20 fim
21 repita
22 |  S' ← Perturbacao(S);
23 |  S' ← BuscaLocal(S');
24 |  S ← CriterioDeAceitacao(S, S');
25 |  S* ← AtualizaMelhorSolucao(S, S*);
26 |  S ← AtualizaMelhorSolucaoNaIteracao(S, S);
27 até CriterioDeEstabilizacao ;
28 se FASE == GRASP então
29 |  MandaSolucaoListaLocal(S*);
30 fim
31 senão
32 |  MandaSolucaoParaPosicaoEspecificaListaLocal(S*);
33 fim
```

Algoritmo 7: Pseudo-código do provedor de solução do AUDImTT.

Nesta tese foram feitos vários testes para calibrar o parâmetro Q, que decide qual a probabilidade de cada processo provedor iniciar da fase de construção GRASP ou da fase ILS, os melhores resultados foram alcançados com $Q = 10\%$. Ou seja, a maior probabilidade é dada para que os processo iniciem da fase GRASP, pois assim as soluções armazenadas na lista de intensificação são renovadas com maior freqüência.

As linhas 17, 18 e 19 do algoritmo são executadas apenas pelos processos provedores que iniciarem da fase de construção GRASP. Em seguida, da linha 21 até 27 é realizada a fase de busca ILS. Essas linhas são executadas por todos os provedores de solução, independentemente de qual fase eles iniciaram. Ao término da fase de busca ILS, a melhor solução alcançada por cada processo provedor é enviada ao gerenciador da lista de intensificação.

Se o processo tiver iniciado da fase de construção GRASP, linha 28, ele enviará sua melhor solução por meio da função `MandaSolucaoListaLocal()` na linha 29. Se o processo provedor tiver iniciado pela fase de busca ILS, ele enviará sua solução através da função `MandaSolucaoParaPosicaoEspecificaListaLocal()` executada na linha 32. Assim, esse processo encerra-se na linha 33 após executar uma iteração completa da fase GRASP e ILS.

6.5.2 Implementação da Aplicação para o Problema da AGMD

A implementação paralela desenvolvida nesta tese para o problema da AGMD, intitulada AUDI-AGMD (*Autonomic and Distributed* AGMD), também foi baseada em uma heurística híbrida GRASP e ILS proposta em [136] e vista na Seção 2.4.

Os pseudo-códigos que implementam os gerenciadores das listas de cooperação e intensificação do AUDI-AGMD são idênticos aos mostrados pelos Algoritmos 5 e 6, respectivamente.

Dessa forma, nota-se que para implementar uma outra heurística paralela através da estratégia proposta, faz-se necessário basicamente implementar o processo provedor de solução e definir as políticas de gerência das listas.

O provedor de solução pode ser o mesmo código desenvolvido para a implementação seqüencial, acrescido das funções de troca de mensagens para que possa haver cooperação entre os demais processos da estrutura hierárquica.

Como adotou-se os mesmos processos gerenciadores das listas de intensificação e cooperação, esta seção descreverá detalhes de implementação apenas dos processos provedores de solução desenvolvidos para resolver o problema da AGMD. O Algoritmo 8 apresenta o pseudo-código desse processo.

Inicialmente, da linha 4 até a linha 9 são inicializadas as variáveis usadas pelo processo provedor. Em seguida, na linha 10 é escolhida a fase em que o processo iniciará, essa escolha é feita por meio da função `EscolheAleatoriamenteFase()`.

Quando a fase escolhida é a fase de busca ILS, linha 11, imediatamente na linha 12 é enviado um pedido de solução para o gerenciador da lista de intensificação. Essa mensagem é enviada pela função `EnviaMsgPedindoSolucao()`. A mensagem de retorno é recebida na linha 13. Essa mensagem pode trazer uma solução ou não.

Caso a mensagem recebida não traga solução alguma, significa que a lista de intensificação estava fazia. Nesse caso, o processo provedor é setado na linha 15 para iniciar da fase de construção GRASP. Se a mensagem trouxer uma solução, a mesma é armazenada nas estruturas locais através da função `AtualizaVariaveisComSolNova()` na linha 18.

A fase de construção GRASP é executada nas linhas de 22 até 26, pelas funções `OTT_M2` e `BuscaLocal()`. Em seguida, nas linhas de 27 até 40, é realizada a fase de busca ILS. Essa busca local poderá ocorrer tanto com uma solução criada pelo próprio processo ou com uma solução de elite enviada a partir da lista de intensificação local ao processo provedor.

Na linha 28 a melhor solução corrente é perturbada por meio da função `Perturbacao()`. Se a condição definida na linha 30 for satisfeita, é aplicada na linha 31 a função de `BuscaLocal()` sobre a solução S'. Em seguida, na linha 32 a função `CriterioAceitacao()` decide se a nova solução substituirá a solução S ou não. A função `AtualizaMelhorSolucao()`, na linha 33 avalia se a solução corrente é melhor do que a melhor solução conhecida pelo algoritmo.

Antes de re-iniciar a busca local, o critério de aceitação é relaxado na linha 36. E a vizinhança é alterada na linha 39. A fase de busca ILS executará até que o *CriterioDeReinicio* seja satisfeito.

Após concluir a fase de busca local, a melhor solução alcançada é enviada ao gerenciador da lista de intensificação. Se o processo provedor de solução tiver iniciado da fase de construção GRASP, a mesma será enviada através da função `MandaSolucaoListaLocal()`, na linha 42. Caso contrário, a solução é enviada pela função `MandaSolucaoParaPosicaoEspecificaListaLocal()` na linha 45.

```
1  Algoritmo: provedor_AUDI − AGMD;
2  Entrada: G = (V, E), CriterioParada, semente;
3  Saída: S*;
4  custo ← ∞;
5  tipo ← DCV;
6  filtro ← [(|V| − 1)/3].M;
7  filtro_bl ← 0,15;
8  CriterioAtualizacaoFiltro ← 100;
9  semente = semente + PegarMyRank();
10 FASE ← EscolheAleatoriamenteFase(Q);
11 se FASE==ILS então
12 |  EnviaMsgPedindoSolucao();
13 |  S = RecebeSolucaoListaLocal();
14 |  se S == NULL então
15 |  |  FASE = GRASP;
16 |  fim
17 |  senão
18 |  |  AtualizaVariaveisComSolNova();
19 |  fim
20 fim
21 se FASE==GRASP então
22 |  repita
23 |  |  S ← OTT_M2(semente);
24 |  até (custo(S) ≤ filtro) ;
25 |  S', S* ← BuscaLocal(S);
26 fim
27 enquanto .NOT.CriterioDeReinicio faça
28 |  S' ← Perturbacao(tipo, S);
29 |  desvio ← (custo(S') − custo*)/custo*;
30 |  se (desvio ≤ filtro_bl) então
31 |  |  S' ← BuscaLocal(S');
32 |  |  S ← CriterioAceitacao(S, S');
33 |  |  S* ← AtualizaMelhorSolucao(S, S*);
34 |  fim
35 |  se CriterioAtualizacaoFiltro então
36 |  |  filtro_bl ← filtro_bl + 0,02;
37 |  fim
38 |  se (tipo = DCV) então tipo ← SAR;
39 |  senão tipo ← DCV;
40 fim
41 se FASE==GRASP então
42 |  MandaSolucaoListaLocal(S*);
43 fim
44 senão
45 |  MandaSolucaoParaPosicaoEspecificaListaLocal(S*);
46 fim
```

Algoritmo 8: Pseudo-código do provedor de solução do AUDI-AGMD.

6.6 Considerações Finais

Este capítulo apresentou as principais contribuições desta tese, que são uma nova estratégia de paralelização para metaheurísticas, um *middleware* para gerenciamento de metaheurísticas em *grids*, e a integração entre os dois.

Com a estratégia de paralelização hierárquica distribuída, uma implementação paralela é dividida em três níveis: gerenciador da lista de cooperação, gerenciador da lista de intensificação e provedor de solução. Os três níveis são independentes entre si, mas trocam informações através da hierarquia de listas de soluções de elite.

Com a estrutura de listas proposta, a estratégia é capaz de implementar naturalmente os processos de intensificação e de intensificação das buscas locais. A primeira é conseguida por meio das várias listas de intensificação implantadas em cada sítio do ambiente. A intensificação sobre as melhores soluções é alcançada pela combinação da lista de cooperação com a operação de renovação.

A hierarquia de listas proporciona que grande parte da comunicação fique centralizada em cada sítio, e que só esporadicamente ocorram trocas de informações fora dos sítios, minimizando o fluxo de mensagens na rede. Outro ponto positivo é que não existe um único processo gerenciando todos os demais, mas sim vários processos gerenciadores, um em cada sítio.

Como conseqüência, todas essas diferenças acima devem ser tratadas para que as metaheurísticas paralelas possam executar eficientemente em um ambiente *grid*. Dessa forma, este trabalho de tese propõe a utilização do *middleware* de gerenciamento metaEasyGrid. Esse *middleware* objetiva transformar automaticamente aplicações paralelas escritas em MPI, em aplicações autonômicas que detêm a capacidade de gerenciarem sua própria execução em ambientes como *grid*.

A integração entre a estratégia proposta e o SGA metaEasyGrid ocorre em nível de compilação, ficando a gerência do ambiente completamente embutida e transparente ao usuário. Além disso, para validar as idéias propostas foram implementadas duas heurísticas paralelas, cujos resultados computacionais são apresentados no Capítulo 7.