UM GRID OPORTUNISTA COM MECANISMO ADAPTÁVEL PELO KERNEL PARA ESCALONAMENTO DE GRÃO FINO

Fábio Henrique Flesch

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Rio de Janeiro
Outubro de 2013

UM GRID OPORTUNISTA COM MECANISMO ADAPTÁVEL PELO KERNEL PARA ESCALONAMENTO DE GRÃO FINO

Fábio Henrique Flesch

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Cláudio Luís de Amorim, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

Prof. Inês de Castro Dutra, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2013

Flesch, Fábio Henrique

Um grid oportunista com mecanismo adaptável pelo Kernel para escalonamento de grão fino /Fábio Henrique Flesch. – Rio de Janeiro: UFRJ/COPPE, 2013.

XIII, 92 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Valmir Carneiro Barbosa

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 86 – 92.

1. Grid Oportunista. 2. Auto-balanceamento. 3. Kernel. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

"Nenhum problema pode ser resolvido pelo mesmo grau de consciência que o gerou."
Albert Einstein

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

UM GRID OPORTUNISTA COM MECANISMO ADAPTÁVEL PELO KERNEL PARA ESCALONAMENTO DE GRÃO FINO

Fábio Henrique Flesch

Outubro/2013

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Aproveitar a energia computacional disponível em workstations e/ou servidores é a principal motivação de grid computing e muitos mecanismos oportunistas têm sido propostos e aplicados nesse sentido. Entretanto, a granularidade das estratégias oportunistas existentes não permitem eficientemente explorar os recursos computacionais disponíveis pelos processos remotos se interação e/ou aplicações frontend estão presentes.

Esse trabalho propõe um *grid* oportunista de *workstations*, denominado *OK*, que através de um mecanismo de limitação(restrição) de recursos possibilita que os *jobs* executem simultaneamente com as aplicações locais em um mesmo nó (*workstation*), sem que aqueles impactem o desempenho dos processos dos usuários. A decisão de quanto e de quando os *jobs* deverão ter seus recursos diminuídos ou aumentados é tomada individualmente por cada nó e de forma dinâmica, levando em consideração apenas a utilização desse ambiente por todos os processos, sejam locais ou não. Uma vez tomada a decisão, esta é posta em prática através de interações do *kernel* sobre os processos originados do *grid*.

Para as funções comuns a todo *grid*, como gerenciamento de fila de *jobs*, encaminhamento e acompanhamento de jobs, utilizou-se uma solução de *middleware* já existente, já em relação aos componentes que caracterizam o *OK* foram realizados alguns ajustes nesse *middleware* e alguns procedimentos foram criados, principalmente na camada de execução.

Finalmente, resultados experimentais com situações reais e artificiais estão presentes nesse trabalho, os quais confirmaram as expectativas quantitativas e qualitativas do grid proposto.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

AN ADAPTIVE KERNEL MECHANISM FOR FINE-GRAINED OPPORTUNISTIC SCHEDULING

Fábio Henrique Flesch

October/2013

Advisors: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Department: Systems Engineering and Computer Science

Taking profit of available computational energy of servers and/or workstations is the main motivation of grid computing and many opportunistic mechanisms have been proposed and employed in this sense. However, the granularity of existing opportunistic grid strategies does not allow for efficient exploitation of available computational resources by remote/grid processes if interactive and/or frontend applications are present. This work introduces an opportunist kernel level mechanism that, through an adaptive mechanism, is able to control dynamic changes of the OS quantum time according to each scheduled thread locality (i.e., local or remote/-grid processes). Consequently, *OK*, the proposed fine grained *O*pportunistic *K*ernel mechanism, is able to offer a high usage of shared processing resources under heavy loads. Experimental results over artificial and production usage of *OK* are presented; qualitative and quantitative expectations are confirmed.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Esse capítulo apresenta as motivações que originaram o presente trabalho, o reconhecimento e a formulação do problema, a partir do qual são então estabelecidos os objetivos pretendidos e as delimitações da pesquisa. Ao final, a organização do documento é apresentada.

1.1 Motivação e Formulação do Problema

A demanda crescente por processamento é uma realidade nos dias atuais, e o principal desafio é atender a essa demanda com o melhor aproveitamento possível dos recursos [1]. Existem diversas abordagens que visam atender esse objetivo, entre estas, destacam-se os *clusters* de processamento [2] e os computadores de grande porte, que são geralmente dedicados. Estes equipamentos, embora eficazes, são custosos, pois serão necessários investimentos não somente com o *hardware*, mas também com o espaço físico, controle de temperatura, energia, processo de instalação, gerenciamento, configuração e manutenção. Essas soluções tendem a ser superestimadas ou mesmo sub-utilizadas, pois, para proporcionar escalabilidade às aplicações que a utilizam e para que possam atender o crescimento natural das demandas por processamento, apresentam configurações maiores que o necessário no momento da aquisição. Logo, pode ocorrer ociosidade de recursos, com o agravante de que quando se tornarem *obsoletas* o ciclo de aquisição se reinicia e, não há garanta que os equipamentos serão reaproveitados.

Uma opção para responder a esse desafio é a adoção da tecnologia de *Grid Computing* [3],que surgiu para suportar o desenvolvimento de diferentes projetos em escala global, portanto, dependendo dos recursos disponíveis e das especifidades técnicas envolvidas, é uma solução viável para abrigar o processamento de aplicações científicas, com a vantagem de que as questões de escalabilidade podem ser tratadas com a inserção ou retirada de nós do ambiente.A infra-estrutura de um *Grid Computing* pode ser classificada como de *serviço* ou *oportunista* [4]. O primeiro foi

desenvolvido para atender aplicações e projetos específicos de organizações que compartilham, através do uso de diferentes *middlewares grids*, uma quantidade *dedicada* de recursos. Esse tipo de *Grid* apresenta, basicamente, as mesmas restrições de investimento que as soluções de *clusters* de processamento [2] e com computadores de grande porte.

Por outro lado, um Grid com infra-estrutura *oportunista* é uma alternativa para obtenção de recursos computacionais por um baixo custo, pois busca utilizar os recursos existentes nos equipamentos dos usuários , que não são dedicados a esse propósito e possuem múltiplas finalidades. Um importante aspecto nesse caso é que os recursos utilizados são os *ociosos*, ou seja, aqueles que não estão sendo utilizados pelos usuários locais, como consequência os processos do *Grid* que executam nas *Workstations não* podem interferir ou impactar nas atividades dos usuários desses equipamentos. Esta limitação faz com que os períodos de utilização fiquem restritos a horários em que não há expediente nas organizações, como, noites e finais de semana, o que faz com que o método de alocação de recursos seja pouco eficiente se comparado com infra-estruturas que dispõem de equipamentos dedicados.

A utilização de um conjunto de workstations em rede para executar ou substituir as funções de super computadores ou clusters dedicados, tem se tornado cada vez mais comum, devido, principalmente, a possibilidade de unir os recursos de cada workstation em um único dispositivo, de forma controlável e escalável. Um dos primeiros trabalhos que sugeriram essa solução foi o NOW — Network of Workstations — [5], o qual demonstra que uma *Network of Workstatons (NOW)* pode atender às demandas de processamento, substituindo e mesmo superando o custo/benefício de super-computadores e sistemas MPP (*Massively Parallel Processor Systems*). Os autores destacam, entre outros, os seguintes desafios:

- *(i):* Diminuir ou eliminar a necessidade de alteração dos sistema operacionais dos nós (workstations) e, principalmente, das aplicações dos usuários que serão executadas;

- *(ii):* Fazer com que a adoção de uma solução NOW traga ganho para todos os usuários – a solução deveria entregar, no mínimo, a mesma performance interativa de uma workstation dedicada, atendendo, através da agregação dos recursos das workstations, as demandas de processamento para os programas dos demais usuários ([5]).

Diversos trabalhos buscam responder a esses desafios com a utilização de diversas estratégias e técnicas. A migração de jobs ([6] [7] [8] [9] [10]), que embora, apresente bom desempenho no que se refere a garantir a *não intrusão (non-intrusiveness)* de jobs oportunistas (isto é, jobs remotos não deveriam impactar os processos dos usuários/*frontend workloads* [11]), apresentam a desvantagem de não serem capazes de

maximizar a utilização dos recursos dos nós. Isso acontece porque ao se realizar a migração nos momentos de picos de utilização, desconsidera-se que esses momentos podem possuir curtos intervalos de duração, e ao término deles o nó voltará a ter recursos disponíveis, tornando essa migração desnecessária. Além disso, ao se realizar uma migração, não há garantia que uma outra migração não será necessária. A fim de contornar essas deficiências, [12] utiliza o recurso de suspensão, ou mesmo finalização de jobs, quando os recursos para processos locais se tornam saturados. Embora esta estratégia não traga impactos significativos na performance aos processos locais, os processos remotos: (i) podem ter um alto custo de *overhead* devido a ocorrência de suspensão/terminação, e; (ii) demandarão o desenvolvimento e/ou adaptação de mecanismos de *fail-recovery* no nível da aplicação.

Virtualização é outra e importante estratégia adotada na computação oportunista [13]. Algumas, como por exemplo [14] executam os processos remotos em máquinas virtuais (MV), estas rodam em background e com baixa prioridade, deixando o sistema operacional responsável por garantir que essa MV não interfira nos processos locais/aplicações *frontend*. Esta solução além de não garantir a *non-intrusiveness*, não permite restringir com granularidade fina a quantidade de recursos que os jobs podem consumir desse nó. Ainda no que se refere a virtualização, outros trabalhos relacionados, como [15], [16], [17] and [18], implementam mecanismos que ligam MV's quando recebem solicitações de processamento de jobs, desligando ou suspendendo as mesmas quando os recursos ficam escassos. Essas soluções embora garantam o não impacto aos usuários dos nós, podem aumentar o impacto causado pelo desligamento das VM's sobre a performance das aplicações remotas, pois o restabelecimento da VM tem alto *overhead*. A estratégia de migração de VM, como explorado em [19], [20], [21], [22], [23], [24] e [11], enfrentam as mesmas restrições que uma migração de job, com agravamento que o overhead envolvido nesse tipo de migração é maior [24].

Outra estratégia é efetuar o processamento dos jobs nos momentos em que os recursos estão completamente ociosos (*Resources completely idling*) [25], i.e., sem nenhum processo local executando. Embora sejam simples e, portanto, alternativas atrativas, requerem o desenvolvimento/adaptação de aplicações específicas para esse tipo de comportamento. Além disso, e o mais importante, com essa granularidade, o potencial dos nós de processamento não é maximizado, pois os usuários, mesmo utilizando os nós, nem sempre estão utilizando todos os recursos do equipamento, portanto esse recurso vago, poderia ser utilizado para processamento de aplicações remotas.

Finalmente, existem estudos que, para restringir recursos a processos oportunistas nas disputas com processos locais, desenvolvem métodos que atuam sobre a maneira que todos os processos são servidos pela CPU(s). Um desses métodos ([26]),

define uma lista de prioridades para todos os processos, alocando cada job com uma prioridade menor que os processos locais. Enquanto [27], busca restringir o acesso a recursos através da definição de um limite máximo (quantidade) de jobs que um nó pode processar. Esses estudos, embora tenham apresentado bons resultados, não fornecem a possibilidade de alocar recursos com ajuste fino [9], pois a quantidade exata de recursos alocados a cada processo não é determinando por nenhum desses métodos. O trabalho apresentado por [28], visa, entre outros, preencher essa limitação ao utilizar de estimativas para definir a quantidade de recursos destinados aos novos jobs. Entretanto, esses ajustes uma vez definidos, podem ser mudados somente no fim da execução do job, i.e., não é possível alterá-los durante a execução do processo.

1.2 Objetivos

Esse trabalho introduz o *OK* (*O*pportunist *K*ernel), um grid oportunista [14] [1] com um mecanismo desenvolvido no nível do *kernel* para ser usado no controle da utilização. A inovação dessa solução fica no nível de granularidade e na forma em que os processos remotos são controlados, pois todas as restrições de recursos são realizadas a nível do *kernel* do nó executor, sendo possível, de forma dinâmica e em tempo de execução, definir a quantidade máxima de recursos que estarão disponíveis para o *grid*.

A quantidade de capacidade de processamento disponível que cada *processos remoto* terá, dependerá da taxa de utilização dos processos locais (que são prioritários). Como essa taxa é alterada constantemente, a alocação de recursos para o(s) *jobs* será(ão) alterada(s) de forma dinâmica e transparente para os usuários do Grid e, principalmente, do nó local. Como o controle está no nível do kernel, as aplicações que rodarão no grid não precisam ser modificadas. O mesmo ocorre com os programas e configurações dos nós executores, pois as modificações feitas no kernel, irão afetar *somente* as aplicações remotas/oportunistas. Além disso, esse mecanismo é autônomo em relação aos demais componentes do grid, pois não há um controlador centralizado determinando qual ação deve ser tomada.

O *OK* é capaz de:

1. Prover um ambiente de processamento *full time*, ou seja, os processamentos ocorrem independentemente do dia e horário;

[1] Nesse trabalho o termo *grid oportunista* ou simplesmente *grid* , será empregado no sentido de utilização dos recursos de processamento ociosos dos nós para a execução de um ou mais jobs, embora nesse trabalho esteja empregado em um ambiente de intranet, o mecanismo pode ser utilizado em um ambiente sem controle centralizado, ficando, portanto de acordo o *checklist* apresentado por [3].

2. Minimizar a interferência dos processos oriundos do grid (também chamados processos remotos, jobs ou processos oportunistas) sobre os processos locais, preservando a performance interativa para usuário;

3. Mudar as restrições de recursos impostas aos processos oportunistas em tempo de execução (*runtime*);

4. Maximizar a utilização dos recursos de processamento ociosos (CPU time) para o processamento de jobs oportunistas;

5. Manter códigos-fontes e configurações das aplicações, que executam nos nós de execução, inalterados.

A capacidade do *OK* é ilustrada pelas três situações (Figuras 1.1, 1.2 e 1.3) a seguir.

A Figura 1.1, mostra uma situação em que um processo local cpu-bound executa sem nenhuma competição por recursos de processamento, nesse caso, o processo local utiliza praticamente toda a capacidade de processamento do nó de execução e demora 3.689 segundos para ser executado. Quando é realizada a inclusão de um processo oportunista, que é idêntico ao local e é iniciado no mesmo instante, observa-se que o processo local utiliza aproximadamente a metade dos recursos de processamento disponíveis. Tendo como consequência a elevação do tempo de processamento para 7.345 segundos, ou seja, o processo oportunista impactou(aumentou) em 99.10% o tempo de execução do processo local, o desempenho desses processos estão mostrados na Figura 1.2.

Ao se repetir a situação anterior no nó, mas com o mecanismo de restrição, proposto nesse trabalho, ativo, percebe-se que o impacto ocasionado pelo processo remoto foi diminuído em 79.27% (de 7.345 para 4.097 segundos). Esse resultado foi obtido pela restrição de recursos de processamento para o processo remoto, quando este está concorrente com o processo local. Caso não exista processo(s) local(is) demandando por recursos, o processo remoto pode aumentar sua fatia de processamento, aproximando-se da capacidade máxima do nó. Dessa forma, o impacto gerado pelo processo remoto é mitigado e os recursos disponíveis são utilizados em sua plenitude, tanto pelos processos locais como pelos processos remotos.

1.3 Delimitação do Escopo da Pesquisa

O escopo do presente trabalho, está delimitado na especificação e desenvolvimento de uma solução baseada em *Grid* oportunista para aplicações que possam conviver com possíveis alterações nos seus tempos de respostas, devido a sua característica de privilegiar os processos locais. Este trabalho não tem por objetivo desenvolver um

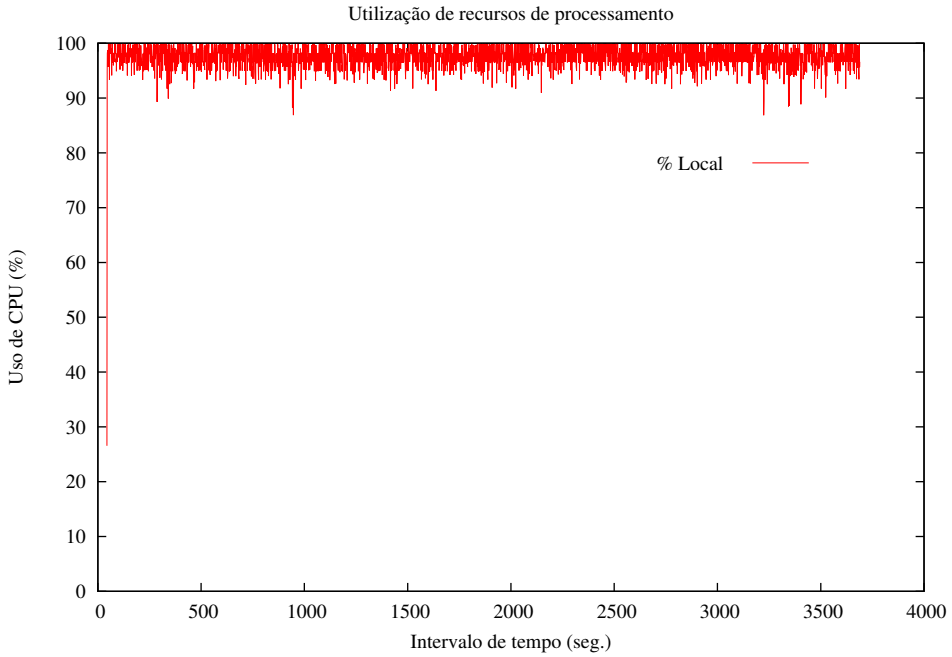

Figura 1.1: Processo local rodando sozinho

middleware [3] , pois o foco está nos nós de execução. Mas para o seu correto funcionamento será necessário um *middleware* com algumas alterações, principalmente na função de escalonamento (alocação do nó para os *processos remotos*).

A granularidade dessa solução está no nível do nó executor, ou seja, os processos oportunistas é que são constantemente monitorados e limitados, podendo ganhar ou perder recursos dinamicamente e sem interferência de um serviço central. Nos períodos prolongados de pico de utilização dos processos locais, uma solução com migração de tarefas poderia ser utilizada, mas deve-se considerar as restrições sobre essa abordagem.

Embora a utilização de uma boa estratégia de alocação de nós seja importante para melhorar o desempenho de um *grid* oportunista, será aplicado nesse trabalho uma política simples, que está baseada nos períodos de não-interação usuário-máquina, ou seja, a prioridade para alocação de *tarefas* será para os nós que estão a mais tempo sem qualquer interação com usuários, a esse tempo é dado o nome de *idletime*. Por limitação do escopo, a implementação de outras estratégias disponíveis na literatura não será efetuada, destas destacam-se [29], [30], [31], [32]. Tais estratégias podem ser incorporadas ao *OK*, possibilitando que o melhor nó executor seja escolhido no momento da submissão de processos oportunista, podendo, inclusive, serem substituídos a medida que outras se mostrarem mais eficientes. Além disso, o *OK* poderia sinalizar situações de carga local para os gerenciadores de recursos, que tomariam decisões de migração ou recusa de jobs.

No que se refere à segurança, o *OK* considera-a no nível de acesso aos dados,

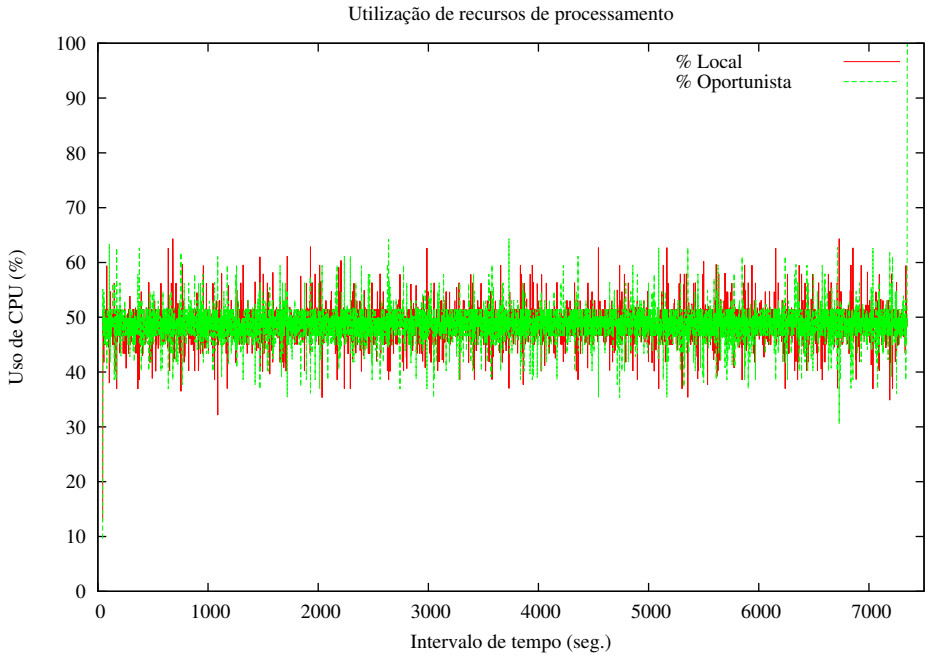

Figura 1.2: Um processo local competindo com um processo remoto com o mecanismo de controle *desativado*

uma vez que dados dos processos não são transferidos nó-a-nó, já que as aplicações armazena-os em um servidor de arquivos e o protocolo NFS *(Network File System)* [33] é utilizado para acessá-los, a proteção fica amarrada às permissões dos usuários, e os acessos são feitos como operações de leitura e escrita em arquivos. Com isso, garante-se que, o usuário, a não ser que tenha acesso ao arquivo em que o processo-grid está atuando, fica impossibilitado de acessá-lo. Isso é possível porque a abrangência dos nós que compõem o *Grid* é limitado a rede local, que permite que a rede não seja um gargalo nas comunicações e o controle de acesso aos dados seja facilitado.

1.4 Organização do Texto

O restante desse trabalho é organizado da seguinte forma: os trabalhos relacionados são cobertos pelo Capítulo 2, o Capítulo 3 apresenta o grid, seus componentes e como é realizada a sua operação. No Capítulo 4, alguns experimentos foram realizados para demonstrar a capacidade do mecanismo, enquanto o Capítulo 5 apresenta a conclusão e os trabalhos futuros.

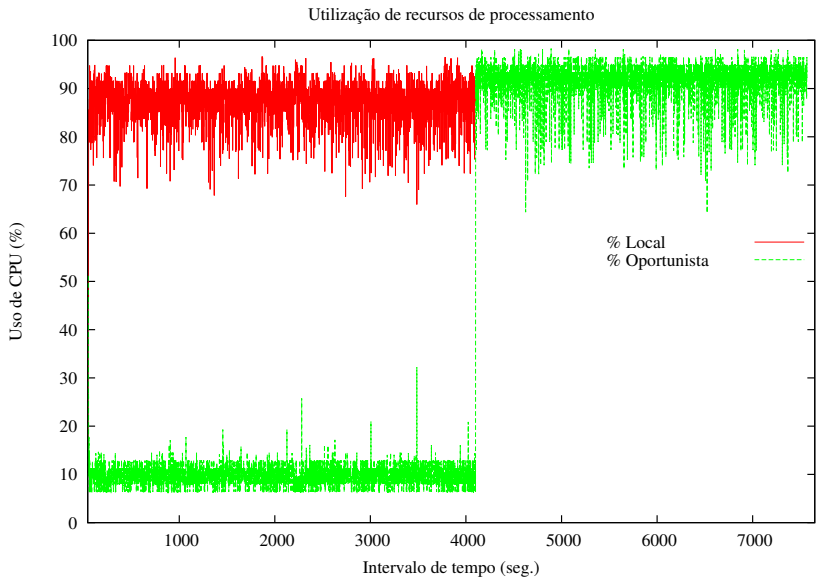

Figura 1.3: Um processo local competindo com um processo remoto com o mecanismo de controle *ativado*

Capítulo 2

Trabalhos Relacionados

Um dos principais desafios que os projetos de grid oportunista ou Desktop grid ([34], [35], [36]), enfrentam é manter a característica de *non-intrusiveness* dos jobs oportunistas, ou seja, os jobs remotos não devem impactar os processos dos usuários (frontend workloads) [11]. Este capítulo apresenta, no que se refere a recursos de processamento, algumas soluções que foram adotadas para enfrentar tal desafio, destacando-se os pontos que diferenciam esses trabalhos do *OK*.

Alguns trabalhos, para controlar as operações e os recursos disponibilizados, se utilizam de *Sandbox*[1] como é o caso da solução Entropia [12], a qual intercepta todas as chamadas API's (*Application Programming Interface*) importantes e realiza as interações com o nó executor, no caso, o *desktop* do usuário. Nas situações em que há um aumento significativo da utilização dos recursos, a *Entropia*, para evitar situações de contenção dos recursos, **paralisa** os *jobs*, e caso essa ação não atenue essa situação, o *job* pode ser **finalizado**. Com essa estratégia, a adoção dessa solução fica restrita, pois quando um job é finalizado, deverá entrar em operação um mecanismo de detecção e recuperação de falha, o que poderá impactar no desempenho final da aplicação, principalmente naquelas em que há restrições de tempo, impactos semelhantes podem ser observados nas situações em que um job é suspenso ou paralisado. O *OK* não necessita de nenhum mecanismo de detecção e recuperação de falhas, pois os jobs nunca são paralisados ou finalizados, já que o controle é feito baseado na diminuição do acesso de recursos a esses jobs.

Outros trabalhos utilizam virtualização [13] para processar os jobs, como é o caso de [15], [16], [17] e [18], que para atender a demanda por processamento ligam a VM nos casos em que há ociosidade e desligam/suspendem quando há falta de recursos. Ao se desligar/suspender uma VM, todos os processos controlados pela VM, são suspensos, o que, além de afetar o tempo de processamento pode diminuir os tipos de aplicações que podem utilizar essa infra-estrutura, já que a suspensão

[1]Mecanismo de segurança para separar os programas em execução, frequentemente utilizado para executar códigos não testados, ou programas não confiáveis

de um job pode ocasionar *timeout*, podendo demandar formas de *fail/recovery* para aquela aplicação. O *OK*, além de ,como citado anteriormente, não necessitar de nenhum mecanismo adicional de controle de falhas, contorna o problema de *timeout* na medida em que a comunicação entre processos não é interrompida, pois o mesmo permanece no estado de execução e o sistema operacional é capaz de tratar as requisições de comunicação com outros processos normalmente, pois um job é um processo normal sob a ótica do SO.

As soluções [19], [20], [21], [22], [23], [24] e [11] optam por migrar/realocar as máquinas virtuais, a fim de contornar as restrições de recursos ou para aumentar/diminuir a quantidade de recursos alocados aos jobs. A relocação de máquinas virtuais incorre em *overhead*, pois para realocar uma VM de um servidor para outro é necessário: *(i)*: salvar o status da VM em execução no servidor original; *(ii)*: transferir essas informações de execução para o novo servidor pela rede; *(iii)*: recuperar a informação de execução no novo servidor; *(iv)*: reiniciar a VM. As operações de gravar as informações, transferi-las e recuperá-las demandam grandes perdas nos tempos de execução [24]. Além disso, não há garantias de que uma segunda, terceira, ou enésima migração não será necessária, o que aumentará ainda mais o tempo de execução. O *OK* contorna essa limitação com a dinamicidade que os ajustes (liberação ou retenção) de recursos acontecem, pois nos momentos de picos de utilização há diminuição do acesso aos recursos por parte dos jobs, na medida que tais recursos são disponibilizados, estes serão alocados aos processos oportunistas

O grid oportunista apresentado por [14], utiliza a estratégia de executar as máquinas virtuais em *background* e com **baixa prioridade**, deixando para o sistema operacional a tarefa de diminuir o impacto causado pelo processo oportunista. Para possibilitar a execução de mais de um tipo de aplicação, foram criadas várias versões de máquinas virtuais (*imagens*) que são instanciadas dependendo da configuração ou tipo do processamento exigido.

Essa solução, embora tenha apresentado bom desempenho, conforme os resultados apurados, não possui garantias, com a granularidade desejada, de que a alocação dos recursos, especificamente do(s) processador(res), não impacte o usuário da máquina hospedeira. Inclusive, não permite definir precisamente os limites máximos que a aplicação oportunista pode utilizar, essas tarefas ficam a cargo do sistema operacional da máquina hospedeira. Esse aspectos são cobertos pelo *OK* pela forma que este aplica seu mecanismo, o qual que além de garantir o não impacto, permite definir a quantidade exata (granularidade) de recursos que serão provisionadas os jobs.

Outro aspecto importante é a complexidade como esse *Grid* trata da questão de múltiplos ambientes de aplicações, pois dependendo do tipo do processamento se faz necessário uma nova máquina virtual, o que pode gerar grandes dificuldades

no que se refere a controle de versões e de armazenamento. O *OK*, permite que independentemente do tipo ou complexidade da aplicação, todas terão o mesmo tratamento e como consequência, uma única instancia por nó.

O Xen [2] é um dos gerenciadores de máquinas virtuais mais relevantes da literatura, sendo capaz de carregar diferentes escalonadores durante a fase de inicialização do domínio hospedeiro como, por exemplo, o EDF *(Earliest Deadline First)* o qual força domínios a executar somente durante uma fatia de tempo a cada período ([37] , [38]). Outros exemplos de gerenciadores incluem o *VMWare*([39]), o *Virtual Box*[40] e o *OpenVZ* , [41]). No entanto, apesar de todos oferecerem a opção de atribuir reservas de processamento as máquinas virtuais, isolar do desempenho de aplicações executando dentro de cada domínio, continua sendo uma tarefa do sistema operacional hospedado na MV. Assim, não há garantia de desempenho entre as aplicações desse domínio [42]. O mecanismo do *OK* atua diretamente no nível do kernel, garantindo que as restrições necessárias (solicitadas) sejam efetivamente cumpridas. O problema de isolamento de desempenho pode ser resolvido atribuindo-se um domínio virtual a cada aplicação, mas dependendo do tipo de virtualização utilizada, essa prática pode ser muito custosa. Assim, em máquinas com muitas aplicações é desejável que existam mecanismos de reserva com granularidade mais fina do que as apresentadas pelas máquinas virtuais, essa granularidade é encontrada no mecanismo do *OK*.

Os trabalhos descritos em [6], [7], [8], [9] e [10] , utilizam técnicas de *migração de jobs* para tentar garantir a *non-intrusiveness* e/ou atender restrições de tempo de aplicações para *Grid* (*Grid Applications*). Para isso, é realizado o monitoramento dos recursos das máquinas(nós) e a disponibilidade de banda de rede, o resultado desse monitoramento é que vai determinar a ação que será tomada, no caso, se migração da(s) tarefa(s) (*task migration*) será(ão) ou não realizada(s). A rotina necessária para realizar a migração, que envolve a transferência do código-executável, dos dados e de todo o contexto do processo, contribui para aumentar *overhead* da aplicação migrada. Em um ambiente com grande utilização dos nós, mesmo tendo rede disponível, a estratégia de migração não é recomendada, pois ao migrar um processo para uma máquina com mais recursos disponíveis em um determinado momento, não garante que esta não se tornará saturada em seguida. Portanto, se um outro nó apresentar melhores recursos um nova migração será necessária, e esses eventos podem continuar acontecendo, o que elevará o tempo de execução desse processo [29]. O *OK* contorna essa limitação com o dinamismo em que as restrições são aplicadas ou retirados dos jobs, pois nos momentos temporários de pico, os jobs têm seus recursos recursos diminuídos, conforme esses momentos passam os recursos são novamente disponibilizados aos processos oportunistas.

[2]Xen hypervisor. http://www.xen.org.

Outro importante trabalho que utiliza a migração de jobs é o *Condor* , nele, os processos oportunistas executam somente quando não há atividade na máquina executora, quando as ações na máquina local são retomadas o job é parado. Caso esse nó executor não esteja disponível em cinco minutos, o job será transferido para outra estação. Essa migração ocorre através através da utilização de *Checkpointing*[43], onde o estado de execução de um programa é salvo antes de ser removido do nó, para que posteriormente possa ser carregado na nova estação executora. Esse trabalho, além do *overhead* relacionado a múltiplas migrações, não potencializa a utilização da capacidade total de um nó, pois a limitação do uso para os momentos em que não há interação na máquina, não permite a execução de jobs nos momentos em que os processos locais demandam poucos recursos de processamento. O *OK* evita que essa situação aconteça, pois as decisões sobre a restrição ou liberação de recursos são tomadas a partir da carga de utilização do sistema naquele momento e não em relação da atividade ou não de um nó

Diversas outras infra-estruturas para *Grid* com auto-ajuste têm sido propostas na literatura [44],[45],[46],[47], todas estas, estão baseados em monitoramento de recursos e migração de tarefas, alguns mecanismos foram inseridos ou modificados nos *middlewares* existentes. Mas todos apresentam as limitações expostas acima devida a estratégia utilizada para o auto-ajuste ser a migração de tarefas.

Para mitigar os efeitos de uma decisão ruim sobre a necessidade de migração de uma tarefa, [29] propõe um mecanismo de reescalonamento em duas fases. Quando o gerenciador de tarefas decide, baseado nas informações coletadas, que é necessário realizar um reescalonamento de uma tarefa, o mecanismo é iniciado; onde a primeira fase consiste em selecionar, via escalonador, outro recurso para continuar a tarefa, essa decisão é baseada na situação atual dos recursos. Antes da decisão ser tomada, é iniciada a segunda etapa que consiste em combinar os recursos em uso atualmente com os recursos do restante do *grid*, caso a capacidade de computação diminua, a realocação não é realizada, caso contrário a migração acontecerá.

O programa *cpulimit* [48] busca realizar a limitação de recursos dinamicamente com o emprego contínuo de pausas e retomadas dos processos controlados pelo cpulimit. Isso é feito através do envio de sinais *sigstop* e *sigcont* para os esses processos. Após alguns testes, esse limitador se mostrou eficiente no que se refere a limitação de recursos, mas não obteve a mesmo desenpenho na atribuição de dinâmica de recursos aos processos. O *OK* não utiliza pausas nos jobs, o controle é feito no tempo de execução dos mesmos. Além disso, alguns processos foram abrotados, provavelmente devido a característica de utilizar diretivas de *sleep* nos processos.

A Linux Kernel Organization [3] introduziu no *kernel* de seu sistema operacional o *Task Control Groups* (TCG ou CGroup), um framework capaz de atribuir compor-

[3]http://www.kernel.org

tamentos arbitrários a determinados grupos de processos com o objetivo de controlar a execução dos mesmos. A ideia de sub-grupos implementada pelo CGroups é semelhante as sub-árvores propostas pelo escalonador hierárquico no trabalho [49]. De acordo com a documentação do kernel Linux, a intenção do *CGroup* é que subsistemas sejam acoplados ao framework para proverem novas funcionalidades tais como a contabilidade e a limita¸ao de consumo de recursos que um grupo de processos pode utilizar. O *CGroup* é capaz de gerenciar de maneira justa os recursos de processamento e disco, e de maneira quantitativa o montante de memoria a ser utilizada por cada grupo de aplicações.

A inserção do *CGroup* no Linux é de grande importância para um maior provimento de qualidade de serviço nesse sistema e expressa a atual preocupação da indústria de software na melhoria do uso de recursos computacionais. No entanto, poucos avanços foram feitos a fim de se prover flexibilidade e extensibilidade às políticas de escalonamento dos grupos de aplicações desse framework. Por esse motivo, a inserção de novas políticas continua sendo uma funcionalidade a ser implementada no código do kernel sistema operacional.

Alguns autores buscam delimitar o acesso a recursos por parte dos processos remotos, atuando sobre a maneira como os jobs são servidos pela cpu(s). Em [26] é definida uma lista de prioridades, onde os processos remotos são classificados com prioridades menores que os processos locais. O trabalho [50] busca controlar o consumo de CPU através da suspensão de jobs, quando os recursos se tornam escassos. Por sua vez, a proposta de [27] define uma quantidade máxima de jobs que cada máquina do grid poderá rodar. Estes estudos apresentaram bons resultados, mas não possibilitam o ajuste fino nas restrições de recursos, pois a carga destinada a cada job é dependente da política de escalonamento do sistema operacional. Uma tentativa de cobrir essa carência está apresentada em [28]. Nesse trabalho, as limitações de recursos aos novos jobs são definidas através de estimativas de utilização dos nós e são implementadas com LXS e Xen. Porém, as limitações impostas a(os) job(s), só poderão ser alteradas após o término do(s) mesmo(s), ou seja, o dinamismo no que se refere a inserir/retirar recursos dos processos remotos em tempo de execução fica comprometido.

Um framework de escalonamento que permite a alteração das políticas de escalonamento sem que modificações nos mecanismos que garantem o compartilhamento do processamento sejam necessárias é encontrado no *RED-LINUX* ([51]). As políticas do *RED-Linux* são implementadas por um processo, que, executando no espaço do usuário, monitora e coleta informações a respeito do montante de recursos necessário para a execução de uma determinada aplicação. Com base no montante estimado, o processo no nível do usuário calcula os parâmetros a serem garantidos e, utilizando uma *API* específica, os envia ao processo despachante. Esse processo

despachante, por sua vez, reside no *kernel* do sistema operacional e é responsável por implementar os mecanismos de garantia de reservas de recursos. Porém, esse trabalho, não permite os ajustes com a granularidade fina desejada (valores estimados) e não garante o não impacto das aplicações remotas sobre os processos locais, pois esses podem aumentar suas demandas a qualquer momento.

Outras iniciativas de gerenciamento de recursos propuseram a implementação de ferramentas no nível do usuário para assegurar reservas de recursos, como é o caso do trabalho descritos por [52], o qual descreve a implementação de ferramentas de reserva de processamento que funcionam utilizando temporizadores os quais, ao expirarem, alteram as prioridades das aplicações sendo executadas em um servidor. Uma dessas ferramentas, denominada DSRT (*Dynamic Soft Real Time Scheduler*), foi projetada para responder as necessidades de aplicações multimídia. Assim, ela permite classificar aplicações de acordo com os seus respectivos perfis de consumo de processamento. O DSRT provê, também, uma API de reserva que pode ser adicionada ao código de aplicações com o objetivo de permitir um controle mais preciso do uso de recursos.

Apesar de ser precursora das ferramentas de reserva de recursos no nível do usuário, sendo parte de projetos de grande relevância na acadêmica,o DSRT teve o seu desenvolvimento descontinuado. Um estudo a respeito das funcionalidades do DSRT mostrou que novos atributos deveriam ser introduzidos na ferramenta a fim de torná-la mais adequada as necessidades de alguns ambientes computacionais atuais. Seria preciso, por exemplo, alterar a forma como os processos clientes e o servidor da ferramenta se comunicam, expandir reservas na presença de processamento ocioso no sistema e prover mecanismos para alocar apenas um subconjunto de unidades de processamento em máquinas com diversos processadores. No entanto, uma análise da ferramenta revelou um código bastante complexo e difícil de ser tratado.

A *sandbox* proposta por [53] monitora o consumo das aplicações para, em seguida, de acordo com o consumo observado, direcioná-las a um comportamento desejado. Para isso, a sandbox utiliza um conjunto de mecanismos disponibilizados pela grande maioria dos sistemas operacionais modernos, tais como temporizadores de granularidade fina, infraestruturas de monitoração, modos de depuração, escalonamento baseado em prioridades e proteção de memória. Essa *sandbox* é implementada para plataformas Windows e impõe restrições quantitativas aos recursos de processamento. Uma desvantagem desse trabalho é que a *sandbox* proposta utiliza o método da interceptação de funções para realizar o controle de uso de recursos, uma técnica que gera uma alta sobrecarga de trabalho [42].

A solução proposta por [42], apresenta uma suíte denominada *ReservationSuite*, que, através da ferramenta chamada *CPUReserve* ([54]), é capaz de limitar o uso de recursos de processamento (inclusive de cpu) através de primitivas providas pelo

sistema operacional *Linux* no nível do usuário, ou seja, sem que instrumentações no *kernel* sejam necessárias. Essa ferramenta conforme os resultados apontaram, obteve excelentes resultados na limitação de recursos de processamento, porém é necessário na *inicialização do nó* informar quantos e quais processadores podem ser utilizados. Dessa forma, para as situações em que nem todos os processadores foram reservados, os recursos dos outros processadores não serão utilizados, mesmo se ociosos. Portanto, além da utilização maximizada dos recursos que pode ficar comprometida, não é possível alterar de forma dinâmica e *on the fly* o número de processadores disponíveis.

A *CPUReserve* gerencia a reserva de processamento através de chamadas ao sistema (*system calls*) para alterar as *prioridades* dos processos. Nesse ponto, o ajuste fino da quantidade de recursos a serem alocados para os processos oportunistas não é exato, pois cabe ao sistema operacional, através do gerenciamento da fila de prioridades determinar a porção de tempo que cada um desses processos terá disponível. Entre as opções de controle dessa ferramenta está a suspensão de job(s), que, conforme citado anteriormente, pode prejudicar o desempenho dos respectivos processos.

Outro ponto interessante dessa solução está na dificuldade em garantir o não impacto dos jobs nos processos locais, pois, os processadores que estão reservados para os jobs deverão concorrer com os processos locais nas situações em que a carga de processamento demandada exija a utilização desses processadores. É nesse ponto que a maior diferença entre o mecanismo proposto e a *CPUReserve* aparece, pois o controle de recursos é maior no primeiro, pois a alteração acontece no nível do núcleo do sistema operacional, e portando sem as mesmas restrições que se aplicam no nível de usuário.

Alguns grids utilizam a estratégia de utilizar os recursos quando os nós estão completamente ociosos (*Resources completely idling*), como é o caso do *Organic Grid ([25]*). Essa solução utiliza os recursos de processamento quando o *screen-saver* de um nó está executando, ou seja, não há usuário utilizando aquele nó naquele momento; quando existe interação de usuário, o job é suspenso e/ou migrado. Essa estratégia, fica restrita, a jobs adaptados a essa situação. Além disso, é importante ressaltar que para um grid oportunista o fato do *screen-saver* não estar ativo não significa que o usuário está utilizando todos os recursos do nó, portanto se existir algum recursos disponível esse deveria ser alocado ao job.

O presente trabalho visa complementar os trabalhos citados anteriormente, proporcionando um mecanismo de controle capaz de limitar com granularidade fina o consumo de CPU em tempo de execução dos jobs (processos oportunistas), sem a necessidade de suspender ou migrar os processos, pois o controle é feito a nível do kernel. Todas as rotinas de controle são realizadas sem a intervenção de um

controlador central, e a implementação das aplicações que rodam no grid permanecem inalteradas, pois todo o funcionamento do mecanismo fica restrito ao sistema operacional.

Capítulo 3

OK: Arquitetura e Implementação

Este capítulo apresenta o projeto e a implementação da arquitetura do *OK*. A seção 3.1 (Projeto) descreve a função de cada camada e a forma como interagem entre si, enquanto a seção 3.2 (Implementação) traz os detalhes de implementação de cada uma delas.

3.1 OK: Projeto

A arquitetura do *OK* está baseada nas fases de execução de uma aplicação em *grid* proposta por [55], e está dividida em três camadas distintas (conforme Figura 3.1): *Camada de Submissão*, *Camada de Alocação* e *Camada de Execução*.

3.1.1 Camada de Submissão

Essa camada é responsável pela identificação dos requisitos mínimos de processamento que a aplicação precisará para executar, sendo acionada quando o usuário invoca a necessidade de processar determinada aplicação em *cluster* ou *grid*, no caso o *OK*. Para atingir esse propósito as seguintes etapas são necessárias:

1. Definição da aplicação: Nessa etapa é verificada se a aplicação solicitada está no escopo de atuação do *OK*;

2. Filtro de requisitos mínimos: Nessa etapa são extraídos os requisitos que são necessários para a execução da aplicação, por exemplo, localização dos dados de entrada e de saída, programas necessários (se for o caso), quantidade de processadores, tipo de processamento e se há requisitos de *QoS*;

3. Verificação\Validação de autorizações: Nesse passo, é verificado se o usuário que invocou a aplicação tem as devidas permissões nos dados que serão manipulados durante a execução da mesma e se este usuário pode realizar esse processamento no *OK*;

4. Enfileiramento do job: Ao passar pelas etapas anteriores, o processo-grid, ou simplesmente *job*, é encaminhado para o *gerenciador de fila*, que por sua vez será o responsável por enfileirar e encaminhar o *job* para execução.

3.1.2 Camada de Alocação

Essa camada é responsável por selecionar (*alocar*), com base nas informações levantadas na camada anterior, qual(is) nó(s) será(ão) utilizado(s) para a referida aplicação. Essa camada possui as seguintes etapas:

1. Coleta e análise de dados: Nessa etapa, os dados sobre a utilização de recursos são coletados e enviados periodicamente pelos nós ao gerenciador de fila;

2. Seleção de nó: Os dados obtidos na etapa anterior são cruzados, pelo *gerenciador de fila*, com os requisitos levantados na seção 3.1.1 a fim de determinar qual(is) nó(s) será(ão) alocado(s) para o processamento;

3. Reserva de recursos: Depois que o nó que efetuará o processamento foi escolhido, é realizada a reserva dos recursos que serão utilizados pelo processo-grid. Essa ação é necessária para evitar que esses mesmos recursos sejam alocados para outro processo gerenciado pelo *Grid*, já que o tempo entre a definição do nó e a efetiva entrada em execução do processo não é previsível;

4. Encaminhamento para execução: Quando todos os requisitos foram atendidos e os recursos foram reservados com êxito, o *job* está pronto para ser encaminhado para a execução.

3.1.3 Camada de Execução

A terceira e última camada mostrada na Figura 3.1 é realizada nos nós que compõem o *OK*, sendo responsável pelo controle da execução do(s) *job(s)*. A divisão para essa camada é feita da seguinte forma:

1. Preparação do *job*: Quando o *job* chega no nó de execução, todos os requisitos de acesso ou dependências relacionadas aos programas e bibliotecas necessários para o seu processamento são verificados, havendo algum conflito, o *job* é sumariamente rejeitado e uma informação com erro é retornada ao *gerenciador de fila*. Senão, o comando com os parâmetros de execução é encaminhado para a etapa seguinte;

2. Execução do *job*: O comando da etapa anterior é realizado, e o respectivo processo é colocado em execução;

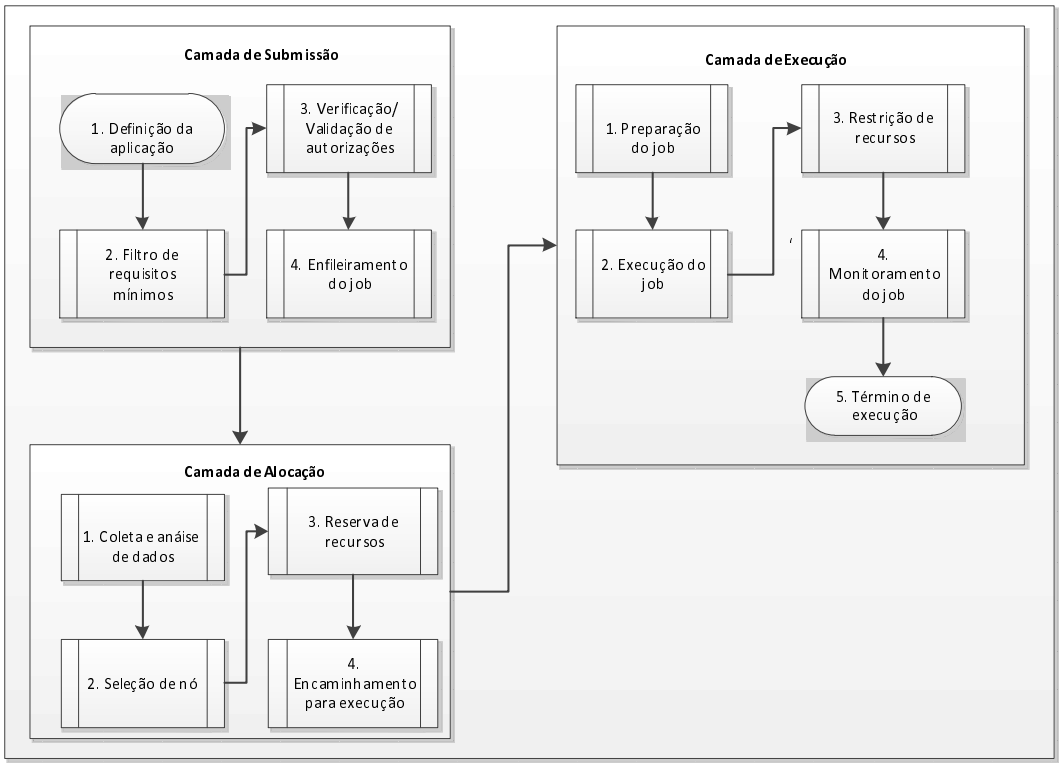

Figura 3.1: Fases de execução de uma aplicação em *grid*.

3. Restrição de recursos: Ao perceber que há um processo-grid em seu ambiente, o nó executor inicia as rotinas de restrição do uso de recursos, tendo por base a disponibilidade de recursos naquele instante;

4. Monitoramento do *job*: O(s) *job(s)* é (são) monitorado(s) constantemente e conforme os recursos utilizados pelos processos locais são liberados ou requisitados, as restrições ao processamento desse(s) *job(s)* também são afetadas, ou seja, se os processos locais demandam recursos, o(s) processo(s) remoto(s) será(ão) limitado(s), caso os recursos sejam disponibilizados pelos processos locais, os mesmos serão alocados aos processos oportunistas;

5. Término de execução: ao término do *job*, ou seja, fim da execução desse processo, as rotinas de monitoramento efetuam as retiradas das restrições que estavam ligadas ao *job* recém terminado.

3.2 OK: Implementação

Nessa seção será feita a descrição da implementação das camadas do *OK*, de acordo com o que foi apresentado anteriormente, na sub-seção 3.2.1 as principais ferramentas auxiliares que foram utilizadas para implementar funcionalidades do *OK* são introduzidas, na sequência é apresentado como as camadas foram implementadas.

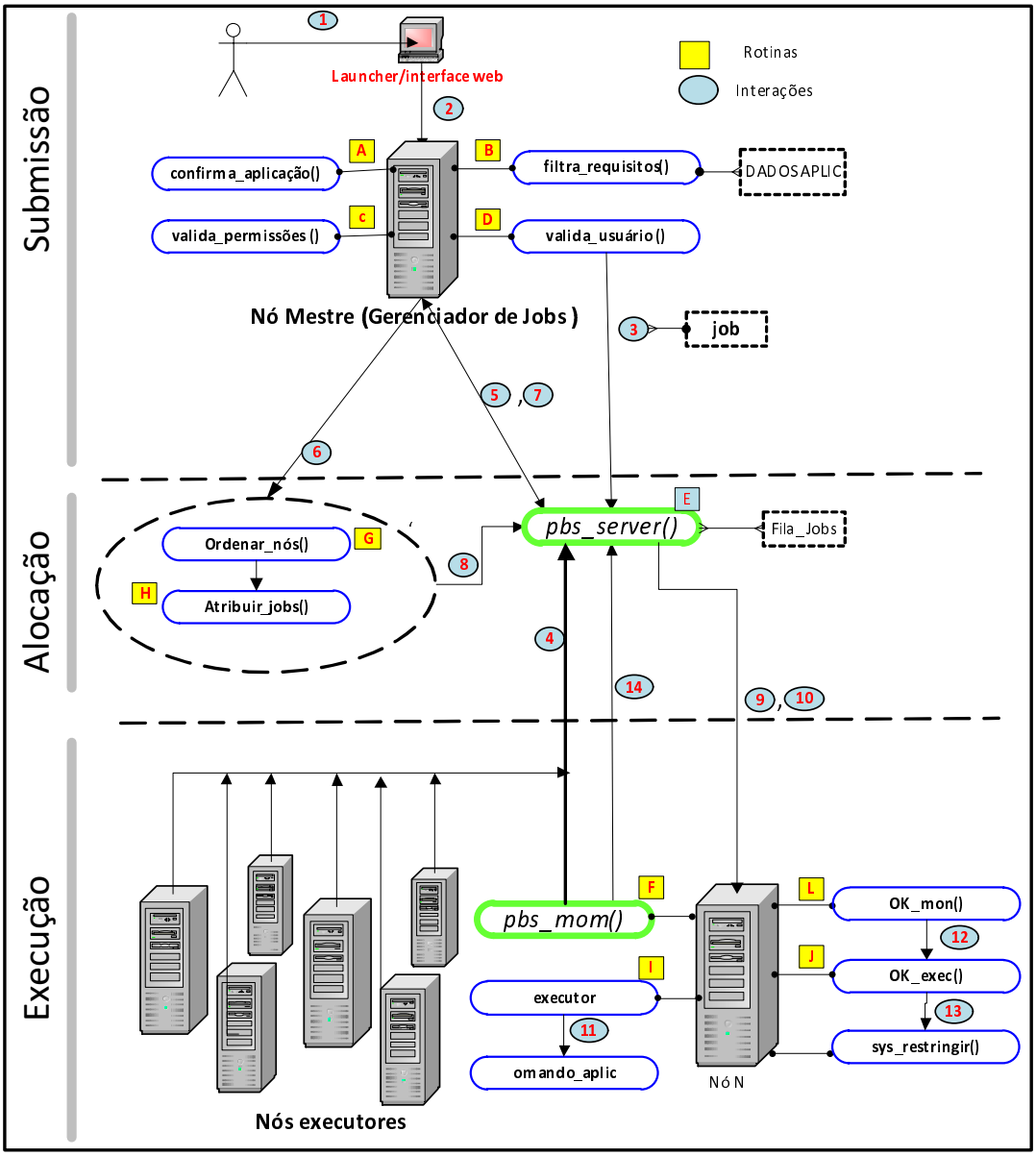

Figura 3.2: Interações de uma aplicação no *OK*.

A Figura 3.2 ilustra as interações entre os componentes do *OK* e a interação deste com o usuário, e será citada nas próximas seções com a intenção de ilustrar o relacionamento entre os componentes do *grid*.

3.2.1 Ferramentas Auxiliares

A quantidade de funções e comandos das ferramentas auxiliares é bem mais ampla do que será mostrado nessa seção, durante o decorrer desse capítulo outros aspectos serão destacados, assim como as alterações que foram necessárias. Para maiores informações recomenda-se a consulta à [56] e [57].

3.2.1.1 Torque Resource Manager

Para atingir seus objetivos, o *OK* necessita de algumas funções comuns a todo *grid* como:

- Gerenciamento da(s) fila(s) de *jobs*;
- Coleta de dados de nós;
- Submissão de *job(s)*;
- Contabilização de utilização.

Para implementar essas funções, foi utilizado, com algumas modificações, o *Torque* (*Terascale Open-Source Resource and QUEue Manager*), que é uma ferramenta de submissão de *jobs* com suporte a *Portable Batch System* ([56]), constituído por quatro componentes fundamentais:

- Nó Mestre (*Master Node*): é o ponto central na comunicação com os nós clientes, responsável por gerenciar a(s) fila(s) de *jobs* e transferi-los para os nós executores ou de computação;
- Nós de Interação /Submissão: Por estes nós os usuários são capazes de submeter e acompanhar seus *jobs*;
- Nós de computação ou de execução: São os nós que processam, monitoram e controlam os *jobs* submetidos;
- Recursos: Representam recursos do sistema e como estão organizados.

Em uma visão geral, um cluster ou *grid* gerenciado pelo *TORQUE* consiste em um *Nó Mestre* e os *Nós de Computação*. O nó mestre executa o daemon *pbs_server* e os nós de computação executam o daemon *pbs_mom*. O nó mestre também tem

um daemon de escalonamento(*scheduler*), que interage com o *pbs_server* no sentido de definir o(s) nó(s) executor(es) para determinado *job*, assim como a quantidade de recursos que será alocada. Não havendo nó disponível o *job permanecerá* na fila, aguardando a liberação de recursos.

Os usuários, através do comando *qsub*, submetem seus processos para o nó mestre (daemon *pbs_server*), que por sua vez, ao receber um novo job contacta o *scheduler* (através do *pbs_server*) para dar prosseguimento a rotina supra-citada de alocação de recursos.

3.2.1.2 Zabbix

O *Zabbix* [57] é uma solução cliente-servidor *open source* para monitoramento de recursos distribuídos, podendo monitorar diversos parâmetros dos equipamentos (no caso nós), como, entre outros: tráfego de rede, utilização de memória e de processador, acessos de entrada e saída. É ainda permitida a definição\criação de novos eventos a serem monitorados. Todas as informações são coletadas por um *agente* e armazenadas em banco de dados, as quais são periodicamente encaminhadas ao servidor para que sejam visualizadas pelas ferramentas gráficas que estão inclusas na solução.

No caso do *OK* a monitoração de cada nó *para a aplicação de restrições* não utiliza o *Zabbix*, e sim um mecanismo local (descentralizado), conforme será visto na seção 3.2.4. Mas por ser um *grid* oportunista é necessário que o desempenho como um todo do *grid* seja acompanhado, e como os processos do *OK* compartilham os recursos de processamento com os processos de usuários locais é de fundamental importância que os recursos dos nós sejam monitorados a fim de verificar e validar a eficácia do mecanismo de compartilhamento. Esse constante monitoramento permite que as estratégias de alocação sejam constantemente revisadas, evitando maiores desgastes, tanto com os usuários locais que não serão impactados, como os usuários do *OK* que poderão ter um desempenho melhor de suas aplicações.

3.2.2 Camada de Submissão

A submissão de um *job*, é iniciada através da interação do usuário com o *OK*, representado pelo **interação 1** da Figura 3.2, que pode ser realizada através de duas formas:

- Console da aplicação (*launcher*): Nesse caso, a aplicação disponibiliza um ambiente em que todos os dados necessários para a submissão da aplicação, no caso um ou vários *jobs*, são extraídos e formatados na sintaxe exigida pelo gerenciador de *jobs*, no caso do *OK* é o *Nó Mestre* do *Torque*. O console é

responsável também por enviar os dados ao gerenciador (através do comando *qsub*).

Essas operações são transparentes para o usuário e esse comportamento é comum na maioria das aplicações que trabalham com *clusters* dedicados ou *grids*, o que varia é o gerenciador utilizado, há diversas opções, como por exemplo, o LSF(Load Sharing Facility) [58] e o OGE (Oracle Grid Engine), conhecido anteriormente como SGE (Sun Grid Engine)[59]. A opção pelo *Torque* foi motivado, principalmente pelo fato de ser uma solução livre e de código aberto com grande aceitação no mercado, o que não descarta a utilização de outros gerenciadores, pois para que se compatibilizem com o *OK*, basta que os procedimentos que foram otimizados para o *Torque* sejam modificados de forma a atender as especifidades de uma ou outra solução de gerenciamento de *jobs*.

- Interface de submissão web: Nos casos em que a aplicação não disponibiliza um console capaz de submeter os seus *jobs*, ou o usuário não possui acesso a um console, essa interface web pode ser utilizada, pois pode ser acessada através de um navegador (*browser*) web. Essa interface deve ser customizada para substituir todas, ou pelo menos as mais importantes, atribuições que um console de uma aplicação deve ter.

  O *OK* possui uma interface que possibilita que os usuários encaminhem seus *jobs* para execução, sendo necessário o preenchimento dos campos seja feito manualmente pelo usuário. Da mesma forma que o console da aplicação, o gerenciador de *jobs* escolhido foi o *Torque*, e a utilização de outros gerenciadores é possível mediante o mapeamento das primitivas utilizadas. A Figura 3.3, mostra essa interface web, embora alguns ajustes no que se referem a segurança e de leiaute devem ser feitos, as funcionalidades que garantem a interoperabilidade usuário-gerenciador de *jobs* estão operacionais.

Ao receber do *console da aplicação* ou da *interface web* uma requisição de processamento (**interação 2** da Figura 3.2), o gerenciador de *jobs* inicia as etapas referentes a *camada de Submissão* (seção 3.1.1), onde cada *job* possui um identificador numérico único, denominado de *jobid*. Algumas alterações nas funções do *Nó Mestre* do Torque, assim como novos procedimentos foram criados, de forma a possibilitar que todas essas etapas fossem realizadas com êxito, no decorrer do texto essas modificações são descritas.

1. Definição da aplicação: Antes de permitir que uma aplicação possa submeter *jobs* ao *OK* é necessário informar (configurar) ao gerenciador de *jobs* que o ambiente está preparado para execução da mesma, ou seja, que o mecanismo de auto-ajuste está preparado para aplicar restrições, se necessário, de acesso a

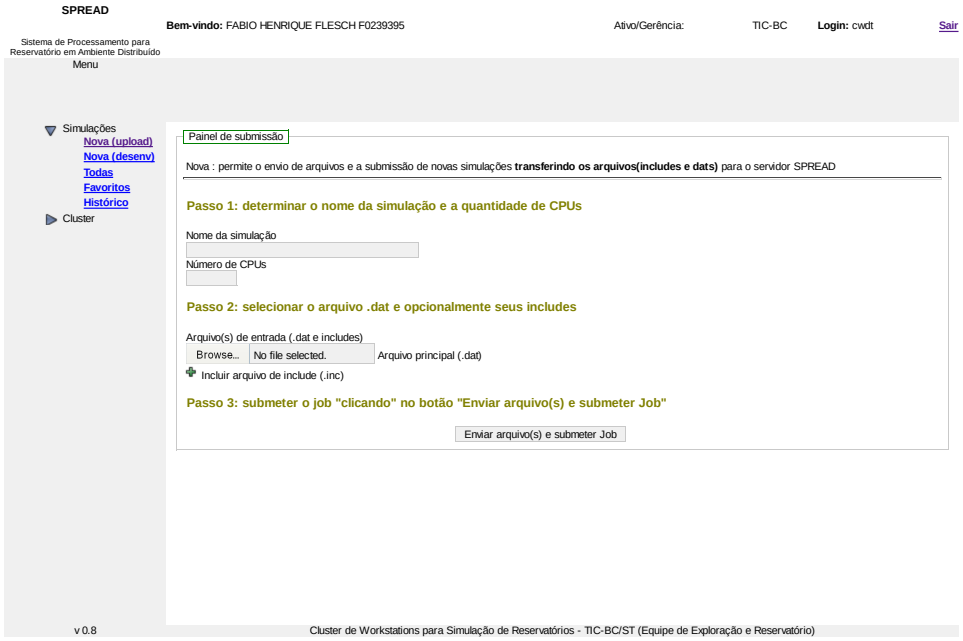

Figura 3.3: Interface de submissão web do *OK*.

recursos de processamento. Portanto, se o *OK* está preparado para receber *jobs* dessa aplicação, a submissão pode continuar para a próxima etapa. Essa rotina é realizada pelo procedimento *confirma_aplicação()* (**rotina A** da Figura 3.2), implementada no gerenciador de *jobs*;

2. Filtro de requisitos mínimos: Ao verificar que o(s) referido(s) *job(s)* faz(em) parte do escopo de atuação do *OK*, as informações repassadas(pelo *console da aplicação* ou da *interface web*) são analisadas pela rotina *filtra_requisitos()* (**rotina B** da Figura 3.2) e apresentam os seguintes dados de retorno:

   - Dados do usuário solicitante: identificação do usuário (*userid*);
   - Aplicação que executará o *job*: Executável com o caminho (*path*) da aplicação;
   - Parâmetros da aplicação, se for o caso: algumas aplicações necessitam, por exemplo, informar a quantidade de *threads* que serão executadas em cada nó, a localização dos arquivos de log, e assim por diante;
   - Localização dos dados de entrada: caminho (*path*) completo da localização dos dados de entrada para o processamento;
   - Local de destino: caminho (*path*) completo da localização em que o resultado do processamento será armazenado;

3. Verificação\Validação de autorizações: É nessa etapa que são verificadas as permissões do usuário que encaminhou o(s) *job(s)*(*função valida_permissões()*, representada por **rotina C** da Figura 3.2) e os locais de origem e destino dos dados. No *OK*, todos os dados, inclusive os executáveis, são disponibilizados através do *NFS*, ou seja, não há transferência de dados na solicitação de processamento, e sim indicação de onde estão os dados e o(s) executável(is). Portanto, caso o usuário solicitante não tenha as permissões de acesso aos dados que são gerenciados pelo *NFS*, a execução desse(s) *job(s)* será negada.

   A validação de usuários também deve ser realizada (*função valida_usuário()* representada pela **rotina D** da Figura 3.2), ou seja, o *OK* possui uma lista de usuários que estão aptos a utilizá-lo, pois, embora a restrição de acesso ao dados das aplicações sejam providos pelos mecanismos de segurança do *NFS*, deve-se evitar que usuários não cadastrados tentem submeter tarefas ao gerenciador, pois poderão causar uma sobre-carga desnecessária, e por consequência, afetando a performance do *OK*.

4. Enfileiramento do job: Ao término das etapas anteriores, o(s) *job(s)* é(são) finalmente encaminhado(s) ao daemon *pbs_server* (**rotina E** da Figura 3.2), para que este inicie o processo de alocação de nó(s) para esse(s) *job(s)*. Essa ação está representada pelo **interação 3** da Figura 3.2.

3.2.3 Camada de Alocação

1. Coleta de dados: Para essa etapa, com exceção dos campos *NP* e *NCP*, não foi necessário a implementação de novas funções ou a alteração de procedimentos do *Torque*, já que os daemon *pbs_mon* (**rotina F** da Figura 3.2) nos *Nós de Execução*, realizam periodicamente, mediante configuração, a coleta de vários dados. No caso do *OK* as informações requisitadas foram:

   - Utilização de processador;
   - Quantidade de memória livre e em uso;
   - Tempo sem interação com usuário (*idle-time*): esse campo informa, em segundos, o tempo sem interação de usuário direta (através de teclado e\ou mouse) nesse nó. Essa informação será utilizada futuramente no mecanismo de seleção do nó;
   - Carga de processamento (*CP*), nesse caso, cada nó de execução tem um valor máximo definido. No caso do *OK*, esse valor depende da quantidade de núcleos de processamento que cada nó possui, exemplificando, caso existam quatro núcleos, a carga de processamento máximo é de quatro.

Essa informação é utilizada para definir se um nó de execução está apto ou não para receber novos *jobs*. Caso esteja apto esse nó é considerado *elegível* para seleção e o status desse nó é definido como *LIVRE*, caso contrário o status fica com o *OCUPADO*, e será expurgado da lista de nós passiveis de receber processamento, até que o valor seja menor que a carga máxima configurada;

- Número de cores por processo oportunista (*NCP*): indica quais jobs, baseado na quantidade de *threads* (qtdade_threads) solicitadas, podem ser executados nesse nó.
- Número de *processos-grid* simultâneos (*NP*): Essa informação indica a quantidade máxima permitida de *jobs*, oriundas do *OK*, que podem executar simultaneamente. Esse limite, configurado e atualizado dinamicamente em cada nó executor, depende da quantidade de núcleos presentes em cada nó e do total de recursos já alocados aos processos remotos, sendo calculado da seguinte forma:
  i. Se não há nenhum processo remoto no nó e este tenha status de *LIVRE*, o valor de *NP* será igual ao número de núcleos de processamento, identificado por *num_núcleos*;
  ii. Ao receber o *primeiro* job, o nó verifica se a quantidade de *threads* (qtdade_threads) demandada é menor ou igual ao valor de *num_núcleos*, em caso positivo, o valor inicial de *NP* é dado pela divisão inteira de num_núcleos por qtdade_threads. Em caso negativo, esse nó não está apto a receber esse job. Além de limitar a quantidade de jobs que esse nó pode receber, o valor de qtdade_threads do primeiro processo remoto determinará que serão aceitos apenas jobs com solicitação de threads iguais a demanda do primeiro processo oportunista (valor de *NCP*).
  iii. A cada novo job recebido, que tenha o mesmo valor de qtdade_threads do primeiro processo-grid, o valor de *NP* é diminuído em um. Esse processo é repetido até que o valor de NP seja igual a zero (indicando que o nó está na capacidade máxima), ou o status para esse nó será igual a *OCUPADO*.
  iv. Quando um job termina sua execução, o valor de *NP* é incrementado em um. Caso não exista nenhum processo remoto em execução no nó, o valor de *NP* volta a ser igual a num_núcleos e esse laço volta ao início.

Por sua vez, o *Nó Mestre* através do daemon *pbs_server*, coordena a obtenção dos dados coletados pelos nós executores (**interação 4** da Figura 3.2). Essas

informações podem ser visualizadas com o comando: *qstat -a*

2. Seleção de nó: Nessa etapa os dados recebidos através da *console da aplicação* ou da *interface web* são cruzados com as informações coletadas pelo nós de execução e consolidadas pelo nó mestre (gerenciador de filas) (**interação 5**), de maneira a definir qual(is) nó(s) de execução receberá(ão) o(s) *job(s)* enfileirado(s). Essa ação é realizada pela combinação dos algoritmos representados nos Algoritmos 1 e 2 (**interação 6 e rotinas G e H** da Figura 3.2).

   O primeiro algoritmo, recebe como entrada a lista dos nós executores (*Lista_nós[]* com os respectivos dados coletados (*idle-time*, NP, CP, memória, NCP, etc). A saída do algoritmo é uma lista dos nós com o status de *LIVRE* ordenados do maior para o menor *idle-time* (*Lista_Elegível[]*). Caso o retorno seja uma lista vazia, conclui-se que não há nós de execução disponíveis e os *jobs* deverão permanecer enfileirados.

Algoritmo 1 Algoritmo de ordenação de nós no *OK*.

Entrada: *Lista dos nós de execução*(*Lista_nós*[])
Saída: *Lista ordenada dos nós a serem alocados para o*(*s*) *job(s)* (*Lista_Elegível*[]).

```
1: nodes_all[] = Ordenar_por_idletime(Lista_nós[]);
2: j = 0;
3: for all nodes_all[i] do
4:    if nodes_all[i].CP = LIVRE then
5:       Lista_Elegível[j++] = nodes_all[i]
6:    end if
7: end for
```

   Esse algoritmo é executado com a periodicidade de 60 (sessenta) segundos, ou seja, a lista de nós disponíveis é atualizada a cada minuto. Esse valor foi definido heuristicamente, através da observação que a execução de diversas interações entre os nó mestre com os nós de execução para se ter os dados em tempo-real, pode provocar queda de desempenho no nó de execução. Infelizmente, não foi identificado, por enquanto, a razão dessa perda de desempenho, portanto será necessário aplicar esforços para corrigir essa situação, caso a solução apresentada (lista ordenada periodicamente e não em tempo-real) apresente baixo desempenho.

   O gerenciador de filas, periodicamente, verifica se há algum *job* enfileirado (**interação 7** da Figura 3.2), em caso afirmativo, a lista gerada pelo Algoritmo 1 é consultada de forma determinar qual nó atende aos requisitos de cada um dos *jobs* que estão na fila, essa ação está representada pelo Algoritmo 2. Nesse algoritmo, para cada *job* enfileirado (*Fila_Jobs[i]*), o gerenciador de filas percorre a lista dos nós executores disponíveis a fim de determinar qual

Algoritmo 2 Algoritmo de atribuição de *jobs*.

Entrada: Lista dos nós de execução(Lista_Elegível[]), fila de *jobs*($Fila\_jobs$[]) com os parâmetros da aplicação

Saída: Relação de execução *job* X nó. ($Lista\_Elegível$[] atualizada)

```
1: for all Fila_jobs i do
2:   for all Lista_Elegível[j] do
3:     if Fila_jobs[i].requisitos = Lista_Elegível[j].requisitos) then
4:       Lista_Elegível[j].jobid ⟵ Fila_jobs[i].jobid
5:       ajusta_status(Lista_Elegível[j])
6:       calcula(Lista_Elegível[j].NP))
7:       atualiza(Lista_Elegível[j].NCP))
8:     end if
9:   end for
10: end for
```

nó possui os requisitos (seção 3.1.1) de processamento solicitados. Caso não exista nó que atenda aos requisitos, esse *job* permanecerá na fila.

Ao final dessa etapa haverá um mapeamento da seguinte forma:

Nó X recebe job N (jobid).

3. Reserva de recursos: Nessa etapa, a partir do mapeamento anterior, o *gerenciador de fila* realiza (através da daemon *pbs_server*) as alterações, se for o caso, nos valores de *CP*, *NCP* e *NP* de cada nó executor; de forma a refletir a solicitação do respectivo *job* (**interações 8 e 9** da Figura 3.2).

4. Encaminhamento para execução: Quando todos os requisitos foram atendidos e os recursos tendo sido reservados com êxito, o *job* está pronto para ser submetido. É função do *gerenciador de filas*, via *pbs_server*, realizar esse procedimento (**interação 10** da Figura 3.2), através do seguinte comando:

/usr/local/bin/qrun -H <noexecutor> <jobid>

Onde, "/usr/local/bin/qrun" é o comando nativo do *Torque* responsável pela submissão de *jobs* aos nós executores, a opção "-H " indica o nome do nó executor (<noexecutor>), e o último parâmetro indica o *job*.

Os parâmetros dos *jobs* para submissão estão em um arquivo, com a seguinte sintaxe de nomenclatura :

<Jobid>.<Nó Mestre>.SC.

É importante frisar que o *Jobid* do comando (/usr/local/bin/qrun -H <noexecutor> <jobid>) é o mesmo do arquivo (<Jobid>.<Nó Mestre>.SC), logo, ao executar o comando, o Nó Mestre e o nó executor interagem, respectivamente, através das *daemons pbs_server e pbs_mom*, e o conteúdo do arquivo será executado pelo nó executor.

O arquivo <Jobid>.<Nó Mestre>.SC contém as seguinte informações:

- Script ou comando executante: Esse campo indica qual *script* ou programa no nó executor irá interpretar as campos abaixo.
- Dados do usuário solicitante: identificação do usuário (*userid*);
- Aplicação que executará o *job*: nome do arquivo responsável pela execução propriamente dita do *job*;
- Parâmetros da aplicação, se for o caso: Quantidade de processadores (número de *threads simultâneas*) e requisitos de *QoS*, embora o último não esteja sendo tratado;
- Localização dos dados de entrada: nome do(s) arquivo(s) de entrada;
- Local de destino em que o resultado do processamento será armazenado: local onde o resultado do processamento será armazenado;

  Onde *"Script ou comando executante"* será denominado *executor* e os demais campos são denominados e tratados como *parâmetros*.

Como todos os dados de entrada são disponibilizados por NFS, não há necessidade de encaminhar para o nó o(s) arquivo(s) de entrada, basta enviar o caminho (*path*) de onde estão armazenados. Já a instalação da aplicação (binário) pode estar instalada localmente ou mesmo mapeada em rede via NFS.

Uma solução com envio e recebimento de arquivos também é possível, desde que haja um *middleware* capaz de suportá-la, no caso do *OK* optou-se por utilizar um ambiente de rede local, pois o princípio fundamental, que é o auto-ajuste nos nós pelo kernel, independe da forma que os dados de entrada e saída são manipulados.

3.2.4 Camada de Execução

1. Preparação do *job*: Ao chegar no nó executor os dados (*Executor* e os *Parâmetros*) do arquivo *<Jobid>.<Nó Mestre>.SC*, serão um comando *UNIX* com a seguinte sintaxe:

   <executor> <Parâmetros>

A execução desse comando resultará no acionamento do *executor* (que pode ser um arquivo binário ou um Script Shell) (**rotina I** da Figura 3.2) tendo todos os parâmetros como argumentos. Dessa forma, é mandatório, que cada aplicação para ser compatível com *OK* tenha um *executor* próprio e instalado em **todos** os nós executores. O Algoritmo 3 traz a descrição das funções básicas do campo *executor*, podendo variar de acordo com o tipo da aplicação e os parâmetros enviados.

Algoritmo 3 Rotina de tratamento de parâmetros pelo *Executor*.

Entrada: *Campo Parâmetros do arquivo <Jobid>.<Nó Mestre>.SC*
Saída: *comando_aplic*

1: $Total\_Esperado\_Parâmetros = Num\_param\_da\_aplicação$
2: $Num\_param = Total\_parâmetros\_recebidos()$
3: **if** $Num\_param = Total\_Esperado\_Parâmetros$ **then**
4: $usuário = P1$
5: $aplicação = averiguar\_compatibilidade(P2, P3, P4, P5)$
6: **if** $aplicação = correto$ **then**
7: $Parâmetros\_aplicação = trata\_requisitos(P8, P9)$
8: $dados\_entrada = formata(P6)$
9: $dados\_saída = formata(P7)$
10: $comando\_aplic = aplicação\ Parâmetros\_aplicação\ dados\_entrada\ dados\_saída$
11: **end if**
12: **else**
13: $ERRO$: *Aplicação não compatível*
14: **end if**

O executor, inicialmente verifica se a quantidade de argumentos é compatível com o que é esperado para iniciar a execução, em seguida, os argumentos são formatados e atribuídos de acordo com a sintaxe exigida pela aplicação e pelo sistema operacional do nó executor, conforme exemplo a seguir:

- *P1*: Identificação do usuário;
- *P2*: Nome do executável;
- *P3*: Versão da aplicação;
- *P4*: Plataforma solicitada;
- *P5*: Local *path* do executável;
- *P6*: Local dos dados de entrada;
- *P7*: Local dos dados de saída;
- *P8*: Parâmetros da aplicação;
- *P9*: Requisitos de *QoS*.

Como as autenticações e validações já foram verificadas anteriormente, pode-se avaliar se a aplicação solicitada (combinação das variáveis *Nome do executável, Versão da aplicação e Plataforma solicitada*) é suportada pelo nó. Em caso positivo, os parâmetros da aplicação e os requisitos de *QoS* (se houver) serão tratados. Se todos os argumentos estiverem corretos, finalmente a execução propriamente dita do *job* pode iniciar, que é realizada com a execução de um comando *UNIX* (*comando_aplic*) com a seguinte sintaxe (**interação 11** da Figura 3.2):

<path_do_arquivo_executável><nome_do_executável> [<Parâmetro 1> <Parâmetro 2> ... <Parâmetro N>]

2. Execução do *job*:

   Nessa etapa, o *comando_aplic* é executado, e o processo criado será tratado normalmente pelo sistema operacional do nó executor, com a diferença que para evitar impactos indesejáveis ao usuário que por ventura estejam utilizando aquele nó, um mecanismo de restrição (descrita a seguir) de recursos é aplicado sobre esse processo (**interação 12** da Figura 3.2).

3. Restrição de recursos:

   O controle e limitação/restrição dos recursos de processamento disponibilizado para cada *job* é feita a partir da implementação de um *daemon*([60]), denominado de *OK_exec* (**rotina J**), que tem, basicamente, a função de executar as ações de aumentar ou diminuir recursos para os *job(s)*, conforme indicação da daemon *OK_mon* (representado pela **rotina L** da Figura 3.2 e descrito na etapa seguinte). A implementação do *OK_exec*() segue o Algoritmo 4.

   O *daemon OK_exec*() (Algoritmo 4), é inicializado junto com os outros *daemons* do sistema operacional e tem como entrada a lista das aplicações que fazem parte do escopo do *OK*, a qual deve ser previamente configurada em todos os nós executores. Além da lista, há os seguintes parâmetros de entrada:

   - Restrição_Inicial (R_0) : indica o valor percentual da restrição inicial que o(s) *job(s)* será(ão) submetido(s);
   - Restrição_Máxima (R_max) : informa a quantidade máxima de recursos de processamento(consumo de CPU) para os processos oportunistas;
   - Ajustar(): rotina para incrementar (UP) ou decrementar (DOWN) o limite de alocação;
   - Restrição_Mínima (R_min) : reporta a quantidade mínima de recursos de processamento(consumo de CPU) para os processos remotos.

Algoritmo 4 *Daemon OK_exec*()

Entrada: *List_Aplic*(*Lista das Aplicações no escopo do OK*)
Saída: *Aplicação da Ação* {Aumentar (UP) ou Diminuir(DOWN) }

```
1: loop
2:   AÇÃO = Escuta_Daemon_OK_mon()
3:   Processos_OK[] = Buscar_Processos_Ativos(List_Aplic)
4:   if Processos_OK[] = ∅ then
5:     break { Não há processos }
6:   end if
7:   if AÇÃO = UP then
8:     for all Processos_OK[[i] do
9:       Limitação_Atual = Limitação(Processos_OK[i])
10:      if Limitação_Atual = ∅ then
11:        Limitação_Atual = 0
12:        Limite_Novo = Restrição_Inicial
13:      else
14:        if Carga_Idle > Restrição_Máxima/2 then
15:          Limite_Novo = Limitação_Atual * 2
16:        else
17:          Limite_Novo = Limitação_Atual + Aumento_padrão
18:        end if
19:        if Limite_Novo > Restrição_Máxima then
20:          Limite_Novo = Restrição_Máxima
21:        end if
22:      end if
23:      Restringir( Processos_OK[i], Limitação_Atual, Limite_Novo )
24:    end for
25:  end if
26:  if AÇÃO = DOWN then
27:    for all Processos_OK[i] do
28:      Limitação_Atual = Limitação(Processos_OK[i])
29:      if Limitação_Atual = ∅ then
30:        Limitação_Atual = 0
31:        Limite_Novo = Restrição_Inicial
32:      else
33:        Limite_Novo = Limitação_Atual/2
34:        if Limite_Novo < Restrição_Mínima then
35:          Limite_Novo = Restrição_Mínima
36:        end if
37:      end if
38:      Restringir ( Processos_OK[i], Limitação_Atual, Limite_Novo )
39:    end for
40:  end if
41: end loop
```

O *OK_exec*() fica em modo de espera até que uma ação encaminhada pelo *OK_mon*() seja recebida. O envio e recebimento das ações são realizados

pela ferramenta *netcat* [61], a qual permite através da manipulação de portas de comunicação TCP/UDP transmitir e receber dados entre nós. A rotina *Escuta_Daemon_OK_mon()* (linha 2 do Algoritmo 4) é a responsável pela recepção e tratamento das mensagens encaminhadas pelo daemon de monitoramento, a sintaxe de utilização da *netcat* é:

nc -l <Porta_de_Comunicação>

Ao receber uma *AÇÃO* ("AUMENTAR" ou "DIMINUIR" os recursos disponíveis), o *OK_exec*(), armazena (via função *Buscar_Processos_Ativos(List_Aplic)*) no vetor *Processos_OK[]* todos os *jobs* que estão executando nesse nó (linha 3), de forma a garantir que todos os processos do escopo do *OK* recebam a mesma ação e por consequência garantam que essa ação refletirá como um todo na alocação de recursos. Esse *daemon*, ao receber uma *ação*, verifica qual o valor atual de restrição dos recursos (variável Limitação_Atual) (linhas 9 e 28), caso não exista (linhas 10 e 29), indica que esse *job* é novo nesse nó e o valor para restrição (*Limite_Novo*) é indicado pelo valor de *Restrição_Inicial* (linhas 12 e 31).

Caso contrário, dependendo da ação recebida, um novo valor é calculado, se for para *AUMENTAR*, o novo limite (Variável Limite_Novo) é o retorno da rotina *Ajustar()* (representado no Algoritmo 4 entre as linhas 13 e 18). A estratégia adotada para essa rotina foi de:

- Nos momentos de grande ociosidade, no caso, Carga_Idle menor que a metade da Restrição_Máxima (linha 14), o novo limite será o dobro da limitação atual (linha 15);
- Para as situações em que há menor ociosidade, a nova limitação será aumentada gradualmente (Aumento_padrão) (linha 17).

Dessa forma, a capacidade de utilização dos recursos, nos momentos com grande ociosidade, por parte dos processos oportunistas é maximizada, pois é possível alocar o máximo permitido de recursos em poucas intervenções. Se fosse adotado apenas aumentos graduais, o número de interações necessário para atingir o pico de alocação seria maior, fazendo com que os recursos ficassem disponíveis por mais tempo, diminuindo a eficiência do *OK*. Se o novo limite superar o valor que é considerado crítico, o novo limite é o valor definido por *Restrição_Máxima* (linha 20).

Caso a ação seja de *DIMINUIR*, o novo limite (Variável Limite_Novo) será equivalente a metade do limite atual do respectivo job (rotina *Ajustar()*, linha

32). Se o resultado for menor que o valor da *Restrição_Mínima*, o novo limite será o valor de *Restrição_Mínima*(rotina *Ajustar()*, linha 35). Com essa configuração é possível em poucas interações restringir ao máximo o acesso a recursos de processamento para os processos oportunistas, mitigando os impactos que estes poderiam causar aos processos locais.

Finalmente, a *System Call*[1] *Restringir(...)* (**interação 13** da Figura 3.2) é invocada (linhas 23 e 38), a qual está associada com a rotina de serviço *sys_-Restringir(processo, limite_atual, novo_limite)* que implementa o algoritmo exibido no Algoritmo 5. Essa rotina recebe como parâmetros de entrada o identificador do processo(*pid*), o valor atual da restrição e a nova limitação a ser imposta ao processo, identificados, respectivamente, pelos campos processo, limite_atual, novo_limite. O novo limite deve ser aplicado, se houver, em todos os *processo-filhos*, através da varredura da lista que contém todos os processos criados pelo processo (linhas 1 a 18 do Algoritmo 5). Essa lista está armazenada no campo "*children*" ([62] ,[63]) do descritor desse processo. Para a continuação do algoritmo, a fila de cada processo_filho (ou simplesmente processo), deve ser bloqueado (*system call this_rq() = lock*), para evitar que ocorra qualquer alteração nos processos alocados nessa fila (linha 2), e antes de iniciar a mudança em algum campo do descritor de processo é necessário desabilitar a preempção (*system call preempt_disable()*) (linha 3).

Com a fila bloqueada e a preempção desabilitada inicia-se a aplicação das restrições solicitadas, ou seja, aumentar ou diminuir os recursos disponíveis para o processamento dos jobs. A estratégia adotada pelo *OK* é interferir no tempo (em *nanosegundos*) remanescente de alocação do processador para cada processo (*quantum ou* time slice) [2] [63]. Essa interferência pode ser de aumentar ou diminuir o tempo, dependendo do resultado subtração dos parâmetros *novo_limite e limite_atual* (linha 8). Se o valor for positivo (novo limite é maior que limite atual), o *time slice* é aumentado percentualmente pelo valor de *Fator_ajuste* (linha 10), em caso negativo, vale o inverso (linha 12). Para as situações em que não há nenhuma restrição (limite atual é igual a zero), o *time slice* é diminuído percentualmente pelo valor de *novo_limite* (linha 6). O novo tempo calculado é atribuído a variável *tempo_OK*, que é o valor que atualizará o campo *time slice* (linha 15). Concluída a mudança no tempo remanescente de processamento, a preempção é habilitada e a fila é liberada (linhas 16 e 17).

[1]Quando um processo em *modo usuário* invoca uma *system call*, o processamento muda para *modo Kernel* e inicia-se a execução de uma função Kernel, que é manipulada através da respectiva *system call handler*. Ao terminar a execução o processamento volta para o *modo usuário*[62]

[2]Essa informação está no campo "time_slice " do descritor do processo

Algoritmo 5 Algoritmo da *System Call* **sys_Restringir()**

Entrada: $processo, limite\_atual, novo\_limite$
Saída: 0 *or* 1 { Sucesso ou falha }

1: **for all** $processo\_filho \in processo$ **do**
2: $fila(processo\_filho) = bloquear$
3: $desabilita\_preempção()$
4: $tempo\_remanescente = processo\_filho \rightarrow time\_slice$
5: **if** $Limite\_Atual = 0$ **then**
6: $tempo\_OK = tempo\_remanescente - novo\_limite$ {Restrição #1, Diminuir percentualmente o tempo }
7: **else**
8: $Fator\_Ajuste = novo\_limite - limite\_atual$
9: **if** $Fator\_Ajuste > 0$ **then**
10: $tempo\_OK = tempo\_remanescente + Fator\_Ajuste$ {Aumentar percentualmente o tempo}
11: **else**
12: $tempo\_OK = tempo\_remanescente - |Fator\_Ajuste|$ {Diminuir percentualmente o tempo}
13: **end if**
14: **end if**
15: $processo \rightarrow time\_slice = tempo\_OK$
16: $habilita\_preempção()$
17: $fila(processo\_filho) = liberar$
18: **end for**

4. Monitoramento do *job*

   O monitoramento é realizado através da implementação de um *daemon* ([60]) denominado *OK_mon* (**rotina L** da Figura 3.2), o qual tem por objetivo avaliar a necessidade de intervenção na disponibilidade de recursos de processamento que está(ão) alocado(s) a(os) *job(s)*. A decisão é tomada com base no nível de ociosidade do nó, esse valor(percentual) é atualizado constantemente e é obtido pela leitura do campo *%IDLE*, que é um dos retornos da execução do comando UNIX *mpstat*(pacote *sysstat*([64])). A essa informação de ociosidade do nó, será dado o nome de *Carga_Idle*. O funcionamento desse *daemon* é apresentado no Algoritmo 6.

   O *OK_mon* é inicializado junto com os outros *daemons* do sistema operacional, e é executado periodicamente (*Tempo_T*) e tem como entrada a lista das aplicações que fazem parte do escopo do *OK*, da mesma forma que o *daemon OK_exec*, esta lista deve ser previamente configurada em todas os nós executores. Além da lista, dois novos parâmetros são utilizados:

   - Carga_minima (L+): Esse valor indica a quantidade percentual mínima de recursos de processamento que deverão estar disponíveis para os pro-

Algoritmo 6 *Daemon OK_mon ()*

Entrada: *List_Aplic*(*Lista das Aplicações no escopo do OK*)
Saída: *Ação a ser executada* { UP ou DOWN }

```
1: loop
2:   if Processo_local ∈ List_Aplic then
3:     CARGA = 100 − Carga_Idle {Utilização do nó}
4:     if CARGA ≤ Carga_Minima then
5:       AÇÃO = AUMENTAR_recursos
6:     else if CARGA ≥ Carga_Crítica then
7:       AÇÃO = DIMINUIR_recursos
8:     else
9:       AÇÃO = Nada a fazer.
10:    end if
11:    Enviar_ação_para_OK_exec(AÇÃO)
12:  end if
13:  Esperar (Tempo_T)
14: end loop
```

cessos *locais*, por exemplo, se *L+=90*, significa que os processos controlados pelo mecanismo só poderão utilizar mais recursos se o percentual de tempo ocioso (*idle*) daquele nó for maior ou igual a 10%.

- Carga_Crítica (L-): Esse limite, indica que a carga de processamento atual desse nó está numa situação considerada crítica, ou seja, há poucos recursos disponíveis para o(s) processo(s) local(is). Logo, deve-se liberar os recursos alocados para os processos remotos. Em uma situação em que *L-=95*, significa que a utilização da capacidade de processamento do nó ao atingir 95% ou mais, fará com que o(s) *job(s)* reduza(m) a utilização de recursos.

Caso exista algum processo que esteja definido na lista de aplicações, ou seja, há pelo menos um *job* em execução naquele nó (linha 3), inicia-se a verificação para determinar se há alguma ação que deva ser tomada em relação a alocação de recursos para o(s) *job(s)*. As ações possíveis são:

(a) Aumentar recursos: permitir que mais recursos sejam alocados para o(s) *job(s)* (linha 5);

(b) Diminuir recursos: retirar recursos alocados para o(s) *job(s)* (linha 7);

(c) Nada a fazer: reservado para uso futuro (linha 9).

Ao término, a ação é encaminhada (linha 11) para o *OK_exec* (**interação 12** da Figura 3.2).

A Figura 3.4 mostra como os recursos de processamento estão divididos (definidos pelas variáveis, L+,L-,R_max e R_mim, mencionadas anteriormente) entre os processos locais e remotos, apresentando a porção que cada grupo de processo pode utilizar e onde poderá ocorrer concorrência . A área *C* (todos os recursos) denota quantidade de recursos que podem ser alocados para os processos locais, embora seja possível existir concorrência com processo(s) remoto(s). A área exclusiva para processos locais é delimitada por *A*. Os recursos para os jobs ficam restritos a área *D*, enquanto a área *F* mostra a porção mínima reservada para estes processos, podendo ocorrer disputa com outros processos em ambos os casos. O pedaço *B* indica a área em que não há atuação do mecanismo e a área *E* denota o range de atuação do mecanismo de controle.

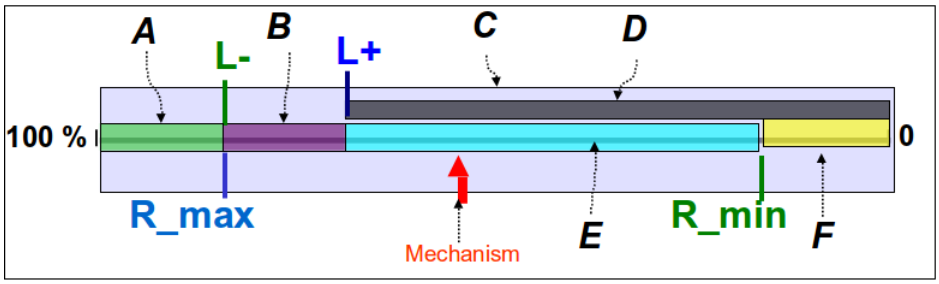

Figura 3.4: Divisão dos recursos de processamento em um nó

5. Término de execução

Ao final do processamento de um *job*, a função de comunicar ao Nó Mestre sobre esse acontecimento, é realizado pelo daemon *pbs_mom*, que informa a alteração do status do *job* de *em execução* para *concluído* ao daemon *pbs_server*, além disso aquele *daemon* atualiza os dados que serão coletados pelo nó Mestre (**interação 14** da Figura 3.2). Nas situações em que é exigido alguma operação pós-processamento, por exemplo, remoção de arquivos de temporários, liberação de recursos, e assim por diante, o *Torque* permite utilização de rotinas para esse fim, que são denominadas de *Epilogue*, de forma análoga existe o *Prologue*, que é responsável pelas ações pré-processamento (maiores detalhes no *Appendix G* do *"TORQUE Administrator's Guide.pdf"* disponível em [56]).

Capítulo 4

Resultados

A fim de verificar o comportamento do mecanismo de controle proposto nesse trabalho, e determinar os momentos que são mais ou menos favoráveis para sua utilização, esse capítulo traz os resultados dos experimentos, incluindo, a descrição de como estes foram realizados. Para isso, foram exploradas, através de aplicações sintéticas e reais, situações possíveis que um processamento em *grid* pode encontrar, sendo analisado o comportamento dos processos (locais ou remotos/oportunistas) nos nós executores com a utilização ou não das restrições aplicadas pela camada de execução desse *grid* (seção 3.2.4).

A verificação do desempenho do mecanismo é realizado através da coleta dos tempos de execução (no nó executor) de cada processo de usuário (processos remotos/jobs e processos locais) duas vezes:

i. Uma vez com o mecanismo desativado, ou seja, os jobs rodam sem nenhuma restrição quanto ao consumo de recursos do processador (concorrendo igualmente por recursos com os processos locais);

ii. Outra vez com o mecanismo ativado, o qual dará prioridade para os processos dos usuários locais, penalizando os processos remotos (jobs).

Com esses dois resultados, pode-se verificar e quantificar a eficiência do mecanismo em limitar a concorrência por recursos dos processos remotos sobre os processos locais dos usuários (que são os prioritários). Nas situações em que o tempo de execução dos processos locais com o mecanismo ativado for *menor* que nas situações em que o mecanismo está desativado, pode-se dizer que o mecanismo foi eficiente, quantificando percentualmente esse ganho. Por outro lado, nas situações em que o tempo for *maior*, é possível afirmar que houve perda de desempenho.

Para a realização dos experimentos, foram utilizadas as seguintes configurações de máquinas:

- Nós de execução: Máquinas com 2(dois) processadores AMD® Opteron™(2,6 GHz/dual-core) (totalizando 4(quatro) núcleos físicos de processamento) e 16 GB de memória DDR-2 667Hz;
- Nó Mestre: Máquina com um processador Intel® Core™Duo (modelo T7500) (2,20 GHz), 2 GB de memória DDR 667Hz;
- Sistema Operacional: O sistema operacional instalado nas máquinas é o CentOS.

A versão do *kernel* utilizada e modificada foi a 2.6.18-348 (presente na versão 5 *update* 9 das seguintes distribuições: *CentOS* [65], *Red Hat Enterprise Linux*[66] e *Scientific Linux*[67] foi lançada em *07/jan/2013*). Embora esta versão não seja a mais atual [1] é considerada a padrão de mercado, por estar homologada para executar a maioria dos aplicativos comerciais utilizados pelos usuários que demandam alto poder de processamento. Porém a migração para a versão 6 é uma necessidade, pois as aplicações estão sendo portadas, e automaticamente os sistemas operacionais devem acompanhar essa migração. Nesse sentido, testes iniciais foram realizados com a versão 6 update 4, que consistiram em substituir e adaptar alguns componentes do kernel dessa versão por componentes desenvolvidos para o mecanismo do *OK*, os testes se mostraram promissores mas há necessidade de realizar a total adaptação do para esse novo *update* ou o encapsulamento do mecanismo em um módulo [62] que possa ser instanciado pelo kernel, conforme está proposto nos trabalhos futuros do próximo capítulo.

A seguir são apresentados os experimentos realizados, com os seus respectivos resultados.

4.1 Experimentos

Todos os experimentos foram realizados com os nós de execução dedicados exclusivamente aos mesmos, ou seja, nenhuma outra aplicação (processo) de usuário estava em execução, portanto, descontando os processos de sistema, cada nó estava com a sua capacidade de processamento (*idle-time)* próximo do máximo. No que que se refere aos parâmetros específicos do *OK*, os seguintes valores foram utilizados:

- Restrição_Inicial (R_0) : 90 %;
- Restrição_Máxima(R_max) : 95 %;

[1] A primeira versão 6 *estável*, para as distribuições mencionadas, foi lançada em *20/jun/2012* (*update* 3) e possui a versão do kernel 2.6.32.279. A versão atual é a 6 *update* 4 (lançada em *21/fev/2013*)

- Restrição_Mínima(R_min) : 10 %;
- Aumento_padrão : 10 %;
- Carga_Mínima (L+): 90 %;
- Carga_Crítica (L-): 95 %;
- Tempo_T: 1 segundo.

A escolha desses valores, embora melhores alternativas devem ser pesquisados (conforme é sugeridos no próximo capítulo), teve como motivação a combinação da premissa de não-impacto aos usuário (*non-intrusiveness*) com o objetivo de maximizar os recursos ociosos dos nós executores. A estratégia adotada teve um viés mais conservador em relação ao primeira, ou seja, entre uma possível maximização da utilização dos recursos e a garantia de não impacto dos processos locais, optou-se por priorizar este. Isso explica a escolha do valor de *L+* ser igual a 90 %, o que na prática, significa que os jobs só poderão utilizar os recursos ociosos se o percentual *idle* daquele nó for maior ou igual a 10%. Com o valor de *L-* igual a 95 %, procura-se garantir uma porção de processamento para os momentos em que os processos locais demandam por mais recursos, ou seja, mesmo com o nó saturado com processos, há uma margem de folga de 5% para os momentos de pico de utilização, caso essa demanda persista, na próximo momento do monitoramento que acontecerá no máximo em 1 segundos (definido por *Tempo_T*.

Essa estratégia em conjunto com rotina *Ajustar()*, permite que nos momentos de grande utilização de recursos os jobs tenham a disponibilidade de recursos diminuída pela metade a cada nova rodado de monitoramento. Essa ação agressiva de redução de recursos permite que em no máximo cinco interações os recursos disponibilizados aos jobs sejam reduzido ao mínimo possível (parâmetro *R_min*), dessa forma os impactos causados pelos jobs aos processos locais são mitigados em no máximo cinco segundos, sendo que no primeiro segundo a proporção de recursos alocados aos processos oportunistas serão reduzidos a metade (conforme descrição na seção 3.1.2 .

Por outro lado, nas situações em que há demanda de processos oportunistas por recursos e esses estão disponíveis, a rotina permite que em poucas interações o valor máximo de recursos definidos (representado pelo parâmetro *R_max*), sejam efetivamentos alocados aos jobs.

Os experimentos foram divididos em três grupos: aplicações *cpu-bound* (sub-seção 4.1.1), aplicações *io-bound* (sub-seção 4.1.2) e aplicação *cpu-bound* real (sub-seção 4.1.3).

4.1.1 Aplicações *CPU-Bound*

Para simular aplicações cpu-bound foi empregado o utilitário *stress* [68], da seguinte forma: esse programa executa isoladamente em um intervalo determinado de tempo, contabilizando-se o número de instruções executadas nesse período (denominado *carga*), desse modo os processos "stress" locais e oportunistas [2] serão finalizados ao atingir esse número de instruções.

A Figura 4.1 representa o comportamento da aplicação, com carga equivalente a 3600 segundos, executando localmente [3], por iniciativa do usuário (sem interação com o *Grid*) e sem concorrer com outro(s) processo(s) de usuário, ou seja, a máquina está executando a aplicação de forma dedicada, portanto, a situação ideal. Os valores do eixo X representam o intervalo de coleta dos dados de utilização de processamento, enquanto o eixo Y representa os valores percentuais de utilização dos processadores, que, para esse processo, ficaram próximos da capacidade total do nó durante os *3.689"*, que foi o tempo total de execução.

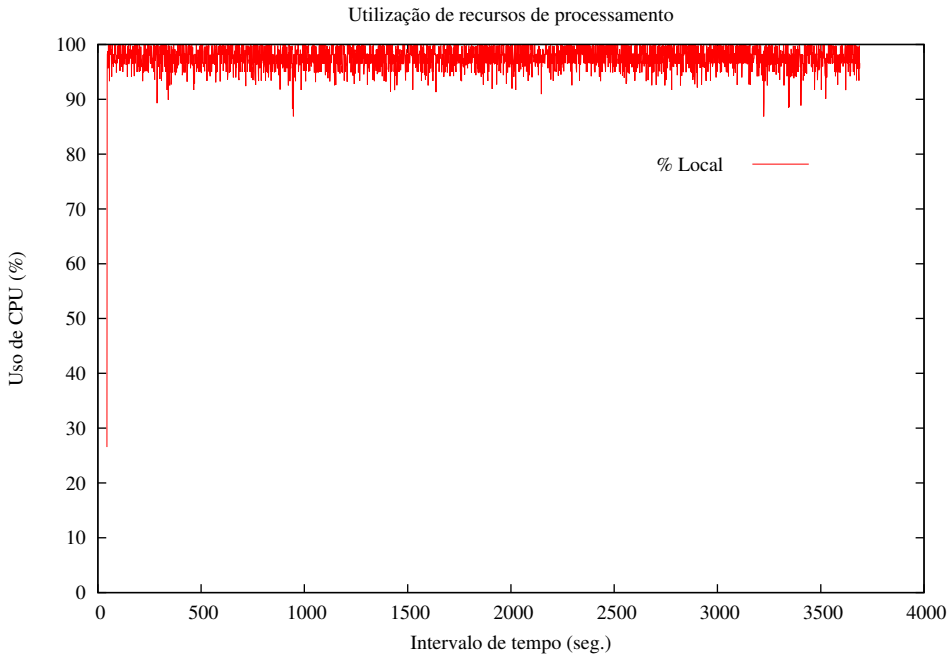

Figura 4.1: Aplicação executando localmente e de forma isolada.

O próximo ensaio mostra o comportamento do nó ao executar, simultaneamente, o mesmo processo anterior (4.1) com um processo oportunista cpu-bound equivalente, ambos com de 3600 segundos, onde:

- Os dois processos foram iniciados ao mesmo tempo;

[2] Nesse trabalho os termos *jobs*, processos oportunistas e processos remotos terão o mesmo significado, denotando o processo oriundo do grid.

[3] Para esse experimento, foram utilizadas 4 (quatro) *threads* que é igual a capacidade máxima do nó executor

- Para evitar algum conflito de *E/S*, os dados de entrada (binários) estão armazenados em arquivos distintos, mas possuem exatamente o mesmo conteúdo;
- O mecanismo de restrição de recursos *não* é utilizado, ou seja, está *inativo*.

Observa-se pela Figura 4.2 que, com o mecanismo inativo, cada processo ocupou aproximadamente a metade dos recursos de processamento desse nó, e os tempos de retorno dos processos local e do remoto foram, respectivamente, *7.345″* e *7.350″*. Ainda em relação ao processo local, o tempo de execução aumentou , quando comparado com o tempo obtido sem concorrência (Figura 4.1). Esse acréscimo foi influenciado pela competição com o processo oportunista por recursos de processamento, pois a luz do sistema operacional ambos os processos são da mesma categoria e recebem o mesmo tratamento. Essa situação ilustra que o processo da aplicação remota, impactou o usuário, pois a tarefa que deveria demorar 3.689″, demorou 7.345″, que é um tempo 99,10% superior ao esperado.

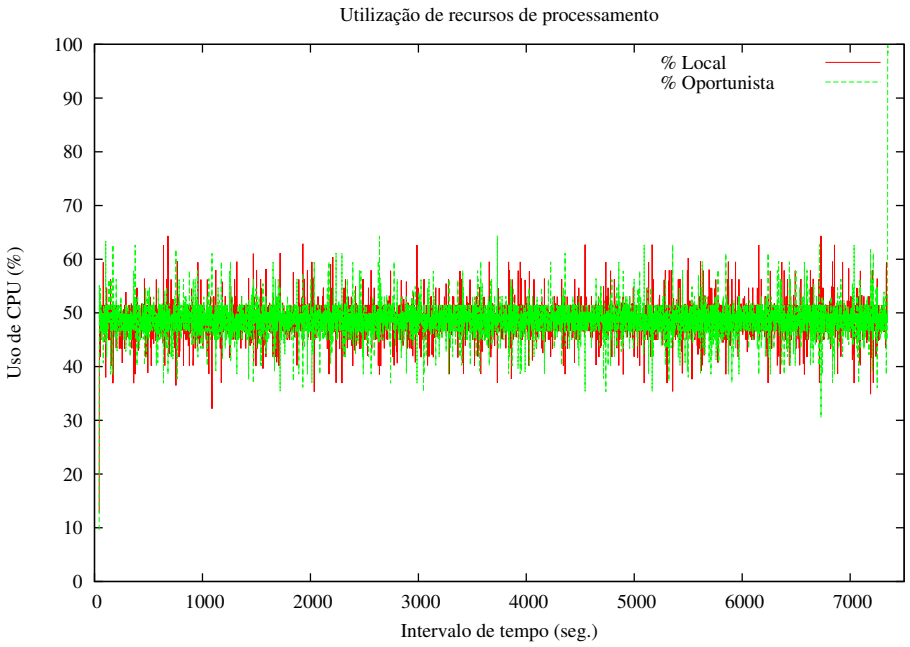

Figura 4.2: Processos local e remoto executando juntos com o mecanismo INATIVO

O ensaio anterior é repetido no próximo experimento (mostrado na Figura 4.3), com a diferença que a execução do *job* é realizada com o mecanismo de restrição de recursos *ativo*. Observa-se que a partir do momento em que o *daemon de monitoramento* identifica que há um processo remoto (intervalo indicado por *(a)* na Figura 4.3), o *daemon de execução* aplica a restrição inicial (intervalo indicado por *(b)*), como a demanda por recursos continua alta, o *daemon de monitoramento* envia uma nova *ação* para o executor, o qual, por sua vez, vai aplicar uma nova restrição. Essa condição permanece constante e só é alterada a partir do momento em que

o *daemon de monitoramento* identifica que o *idle–time* é maior ou igual ao carga mínima ($L+$) (intervalo indicado por *(c)*), que é o instante em que o processo local está sendo finalizado. Nos intervalos seguintes o limite é aumentado até atingir o máximo permitido da disponibilidade do nó, essa condição permanece inalterada até que o *job* finalize sua execução (intervalo indicado por *(d)*).

O tempo de retorno para o processo local ficou em *4.097″*, superior ao tempo considerado ideal, mas melhor que *7.345″* que foi o tempo obtido sem a utilização de restrição no *job*. Por outro lado o desempenho do *job* ficou *7.569″*, que foi superior ao tempo (*7.350″*) registrado quando não havia restrição. Embora o tempo de execução do *job* tenha sido maior, o impacto no processo do usuário que executava naquele nó foi diminuído em 44,22%, ou seja, o mecanismo mitigou o impacto gerado pela concorrência do processo remoto ao processo local.

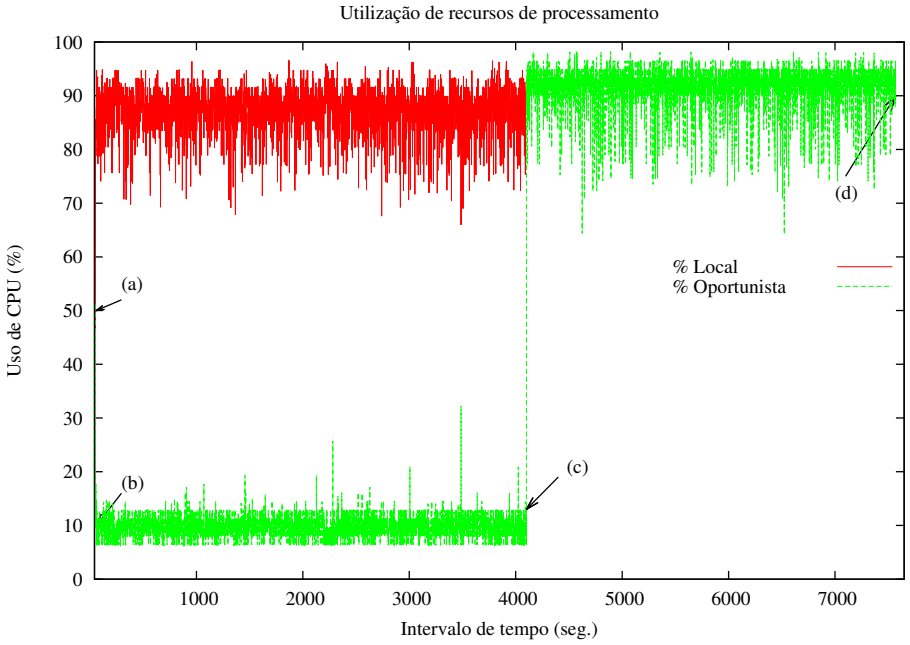

Figura 4.3: Processo local e remoto executando juntos com o mecanismo ATIVO

Os próximos experimentos têm por objetivo mostrar o desempenho do mecanismo nas diversas situações que um nó desse grid pode se encontrar. Para isso, são exploradas desde situações com poucos processos (locais e remotos) e baixa taxa de utilização dos recursos de processamento, até situações em que há um grande número de processos e alta taxa de utilização do referido nó. A definição do momento de chegada de cada processo é estocástica, sendo, então, necessário conhecer a distribuição de probabilidade que descreve os tempos entre chegadas sucessivas (tempos entre chegadas)[69]. Da mesma forma a carga de cada processo (taxa de serviço), não assumira um padrão determinístico, mas sim um padrão aleatório. A obtenção do tempo entre chegadas (λ) e da carga (μ) seguem a metodologia descrita a seguir.

Há uma série de métodos usados para gerar valores para variáveis aleatórias não-uniformes. Cada método é aplicável somente para uma parte da distribuição em questão. Ainda, para uma distribuição particular, um determinado método pode ser mais eficiente do que outros. Descrever-se-á aqui apenas um método, usado para gerar valores para uma variável aleatória que segue uma distribuição exponencial, denominada *Transformação Inversa*.

Este método baseia-se na observação de que, para uma dada uma variável aleatória x, com uma CDF (*Cumulative Distribution Function*) [4] $F(x)$, a variável $\upsilon = F(x)$ é uniformemente distribuída entre 0 e 1. Assim, os valores de x podem ser obtidos gerando-se números aleatórios uniformemente distribuídos e computando-se: $x = F^{-1}(\upsilon)$. A prova desta observação está demonstrada em [70]. Sendo assim, dado um determinado valor x, e uma taxa média de chegada λ, calcula-se a probabilidade de ocorrência de x ou ainda a probabilidade acumulada $P(X \leq x)$. Agora, dada a probabilidade $P(X \leq x) = F(x)$, pode-se calcular o valor correspondente de x usando a transformação inversa usando a transformação inversa:

$$F(x) = 1 - e^{-\lambda.x} = \upsilon \quad \therefore \quad x = -\frac{1}{\lambda} ln(1 - \upsilon) \tag{4.1}$$

Assim, valores para a variável aleatória x_i podem ser produzidos através da geração de uma variável υ_i na Equação anterior . Como υ é uniformemente distribuída entre 0 e 1, a expressão 1 – υ também é distribuída uniformemente entre 0 e 1.

A partir da Equação 4.1, o instante de execução *(t)* de cada processo é calculado isoladamente da seguinte forma:

$$t = -\frac{1}{\lambda} ln(1 - \upsilon) \tag{4.2}$$

A carga *(l)* de cada processo é obtido a partir da Equação 4.1, onde λ (taxa média de chegada) será substituída pela *média de entrega de serviço*, identificada por μ, conforme a Equação 4.3.

$$l = -\frac{1}{\mu} ln(1 - \upsilon) \tag{4.3}$$

A *taxa de utilização* ρ, é uma medida da intensidade que os recursos em um nó estão sendo utilizados e é calculado a partir da relação:

[4] *Cumulative Distribution Function* ou PDF – *Probability Distribution Function*), também chamada função de distribuição, é a probabilidade de X assumir valores menores ou iguais a r, onde r é um número real.

$$\rho = \frac{\lambda}{\mu} \tag{4.4}$$

Quando $\rho > 1$, isto é, $\lambda > \mu$, a taxa de chegadas excede a máxima taxa de serviço do sistema, é esperado, com o passar do tempo, que a fila se torne cada vez maior, a menos que novos usuários não sejam autorizados a entrar no sistema. Se o interesse reside nas condições de estado de equilíbrio (o estado do sistema após estar em operação durante um longo tempo), quando $\rho > 1$, o tamanho da fila nunca se estabiliza e não há estado de equilíbrio. Verifica-se que para os resultados de estado de equilíbrio existirem, ρ precisa ser estritamente menor do que 1. Quando $\rho = 1$, a menos que chegadas e serviço sejam determinísticos e prefeitamente programados, não existe estado de equilíbrio, pois a aleatoriedade vai evitar que a fila nunca se esvazie e permitindo que os nós continuem buscando processos, causando assim um crescimento sem limite na fila. No entanto, conhecendo-se a taxa média de chegada e a taxa média de serviço, pode-se calcular o número mínimo de processos em paralelo para garantir uma solução de estado de equilíbrio encontrando o menor valor de ρ tal que $\frac{\lambda}{\mu} < 1$ [69].

Para a condução dos experimentos, foram adotadas as seguintes premissas:

- O intervalo de tempo para recebimento de processos (locais e remotos) - janela de submissão *(w)* - é de duas horas (7200 segundos), não havendo limitação para término de processamento de cada processo;
- Os processos serão gerados /classificados de acordo com a carga de cada um deles (que pode ser normal ou pesada) e da origem dos mesmos (local ou remoto), totalizando quatro tipos: local-normal, local-pesado, remoto-normal e remoto-pesado.
- O valor da média de chegada dos processos normais é menor que 1.
- A média de chegada dos processos pesados remotos é igual a um terço da janela de submissão *(w)*.
- Os processos remotos normais não serão submetidos ao grid.

Como cada tipo de processo é independente no que se refere a carga e momento de entrada em execução, existe um valor λ e de μ para cada tipo de processo, totalizando 8 variáveis, conforme mostra a Tabela 4.1.

Considerando as premissas citadas anteriormente, os valores de λ_remoto_normal e μ_remoto_normal serão iguais a zero, pois os processos do tipo remoto-normal não serão submetidos. Para os processos remotos-pesados, em

Tabela 4.1: Variáveis μ e λ dos processos locais e remotos

Variáveis locais	Variáveis Remotas
λ_local_normal	λ_remoto_normal
λ_local_pesado	λ_remoto_pesado
μ_local_normal	μ_remoto_normal
μ_local_pesado	μ_remoto_pesado

que a média de chegada dos processos é equivalente à 2400 segundos [5] o valor de λ_remoto_pesado é igual a 1 /2400. Os demais valores de μ serão determinados através da Equação 4.4, citada anteriormente.

As médias de frequência de chegada de processo locais-normais, para os experimentos, foram definidas como baixa, média e alta frequência. As quais são representadas, pelos valores 0,1, 0,5 e 0,9 para a variável λ_local_normal. Enquanto a médias de frequência de chegada de processo locais-pesados é definida em proporção da frequência de chegada de processo remotos-pesados (λ_remoto_pesado), podendo assumir: um valor *igual* ou dez vezes inferior a (λ_remoto_pesado). A Tabela 4.2, mostra como os valores serão distribuídos ou calculados nas variáveis dos λ e μ dos processos.

Tabela 4.2: Atribuições de valores as variáveis μ e λ dos processos locais e remotos

Variáveis locais	Valor	Variáveis Remotas	Valor
λ_local_normal	0,1, 0,5 e 0.9	λ_remoto_normal	0
λ_local_pesado	{1 ou 0,1} λ_remoto_pesado	λ_remoto_pesado	$= \frac{1}{w}$
μ_local_normal	$= \frac{\lambda\_local\_normal}{\rho}$	μ_remoto_normal	0
μ_local_pesado	$= \frac{\lambda\_local\_pesado}{\rho}$	μ_remoto_pesado	$\frac{\lambda\_remoto\_pesado}{\rho}$

Visando cobrir as variações dos valores das variáveis λ_normal_pesado e λ_local_normal, os ensaios serão divididos em dois *lotes*, o primeiro (*Lote-1*) considera a taxa de frequência de chegada dos processos normais pesados dez vezes menor que a taxa de processos remotos-pesados, ou seja, $\lambda\_normal\_pesado = 0{,}1 * \lambda\_remoto\_pesado$. Enquanto o segundo (*Lote-2*) igualará as taxas de chegadas de processos normais e remotos pesados ($\lambda\_normal\_pesado = \lambda\_remoto\_pesado$).

Para a execução de cada lote, foi considerado as situações com baixa, média e alta taxa de utilização dos recursos do nó (ρ), quantificados, respectivamente, por 10%, 50% e 90%. Considerando que será necessário utilizar os valores de λ_local_normal em conjunto com ρ para calcular os valores de μ dos processos e que há três valores

[5] Um terço do valor da janela de submissão *(w)*, ou seja, 7200 /3.

para as taxas de frequência de chegada de processos locais-normais, o resultado dessa combinação será igual a 09 (nove) ensaios ou cenários. São eles:

Cenário 1 : $\lambda\_local\_normal = 0{,}1$ e $\rho = 10\%$

Cenário 2 : $\lambda\_local\_normal = 0{,}1$ e $\rho = 50\%$

Cenário 3 : $\lambda\_local\_normal = 0{,}1$ e $\rho = 90\%$

Cenário 4 : $\lambda\_local\_normal = 0{,}5$ e $\rho = 10\%$

Cenário 5 : $\lambda\_local\_normal = 0{,}5$ e $\rho = 50\%$

Cenário 6 : $\lambda\_local\_normal = 0{,}5$ e $\rho = 90\%$

Cenário 7 : $\lambda\_local\_normal = 0{,}9$ e $\rho = 10\%$

Cenário 8 : $\lambda\_local\_normal = 0{,}9$ e $\rho = 50\%$

Cenário 9 : $\lambda\_local\_normal = 0{,}9$ e $\rho = 90\%$

O ponto de partida para cada cenário é a geração de quatro listas (uma para cada tipo de processo), com o seguinte conteúdo :

- A *carga* de cada um dos processos, calculado através da fórmula 4.3 pelo respectivo valor de μ, conforme mencionado anteriormente.
- O $momento(m)$ em que cada processo deve entrar em execução, que é o somatório dos instantes de execução *(t)*(calculado pela Equação 4.2, utilizando-se o respectivo valor de λ) de todos os processos anteriores daquele tipo (inclusive o processo atual). Quando o somatório superar o valor da janela de submissão*(w)*, nenhum outro processo desse tipo será inserido nessa lista e a mesma será encerrada.

Em seguida é realizada a concatenação e ordenação pelos $momentos(m)$ das quatro listas geradas anteriormente, resultando em uma lista, denominada *lista de entrada*, com a sequência de processos que o nó vai executar. Finalmente, cada lista de entrada é executada pelo nó duas vezes (uma vez com o mecanismo *ativo* e outra com o mecanismo *inativo*), coletando-se o tempo de processamento de cada processo.

Cada lote foi executado por dez vezes com cenários (listas de entradas) diferentes, e os tempos coletados relativos a cada cenário foram concatenados em um único resultado [6]. Realizado esse procedimento, é possível avaliar se o mecanismo propiciou

[6]Cada lote possui 9 cenários, sendo dois lotes, totalizando 18 entradas

ganho de performance para os processos *locais*, ou seja, pode-se determinar de forma quantitativa se o mecanismo mitigou ou não o impacto causado pelos processos remotos aos processos locais. Tal avaliação é realizada através de:

(i) Cálculo das médias dos tempos de execução de todos os processos.

(ii) Distribuição acumulativa dos tempos de execução dos processos *locais* [70];

As duas próximas seções exploram os resultados da execução dos lotes.

4.1.1.1 Resultados Aplicações *CPU-Bound*: Lote - 1

O desempenho do *OK* em todos os cenários, considerando o tempo médio de execução de todos os processos com o mecanismo ativo e inativo, está apresentado na Figura 4.4. Observa-se que nas condições em que a carga de trabalho está baixa ($\rho = 10\%$ - *Cenários 1, 2 e 3*), os ganhos para os processos locais foram pequenos (*colunas 10 - 0,1* e *10 - 0,5*) ou mesmo aconteceram perdas (*coluna 10 - 0,9*). Isso deve-se ao fato de que, como esses processos são relativamente pequenos, o tempo de concorrência com os processos remotos é menor, podendo, inclusive, nem acionar o mecanismo de controle (caso o processo inicie e termine entre o intervalo de monitoramento). Essa condição se agrava quando o valor de λ_local_normal é 0,9, pois o tamanho médio dos processos são ainda menores e o *overhead* gerado pelo mecanismo é maior que os possíveis ganhos de performance. Esse resultado foi confirmado pela verificação das distribuições acumulativas para $\rho = 10\%$ (Figuras 4.5, 4.6 e 4.7), pois verifica-se que o comportamento dos processos locais com o mecanismo ativo e inativo, representados por cada uma das curvas, são praticamente idênticos, caracterizando que não houve ganhos representativos.

Para os cenários em que as taxas de utilização são maiores, aconteceram ganhos de performance em todos os casos, principalmente nas situações em que a frequência de chegada de processos é menor (cenários *4* e *7*). Isso acontece porque os processos locais concorrem pouco entre si, e mais com os processos-oportunistas, e como aqueles possuem uma carga maior (em relação aos processos dos demais cenários), concorrerão por mais tempo com processos remotos. Esse comportamento irá propiciar o acionamento contínuo do mecanismo, resultando na limitação de utilização de recursos para os jobs e consequentemente mitigando o impacto nos processos locais.

A diferença de desempenho para esses cenários (Figura 4.4 - (*colunas 50 - 0,1* e *90, 0,1*), que possuem os mesmos valores de λ_local_normal, é explicado pelo fato que o tamanho médio de cada processo é maior conforme aumenta o valor de ρ, o que gera disputas maiores entre os processos locais. Havendo mais concorrência entre processos locais, os recursos disponibilizados através das restrições aos processos remotos, devido a constante ação do mecanismo, serão alvo de disputa por esses

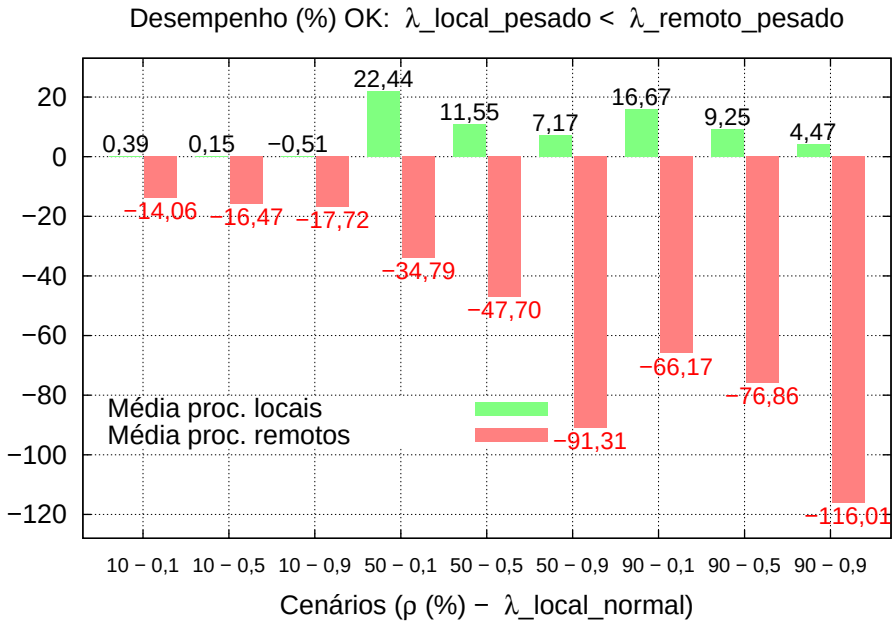

Figura 4.4: Resultado da ação do mecanismo de restrição sobre o tempo médio dos processos.

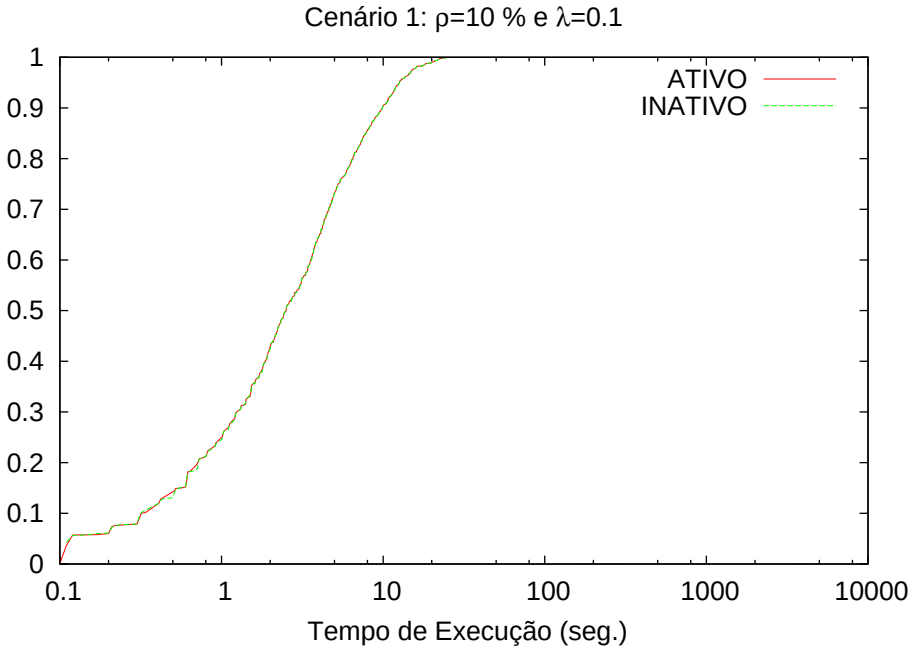

Figura 4.5: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 1*.

processos, diminuindo os ganhos propiciados pelo *OK*. As distribuições acumulativas desses cenários estão presentes nas Figuras 4.8 e 4.14.

Os demais cenários mantêm o comportamento descrito anteriormente, conforme pode-se observar pelo dados presentes na Figura 4.4 - (colunas: *50 - 0,5*, *50 - 0,9*, *90, 0,5* e *90, 0,9*) e as respectivas distribuições acumulativas (Figuras 4.10, 4.12,

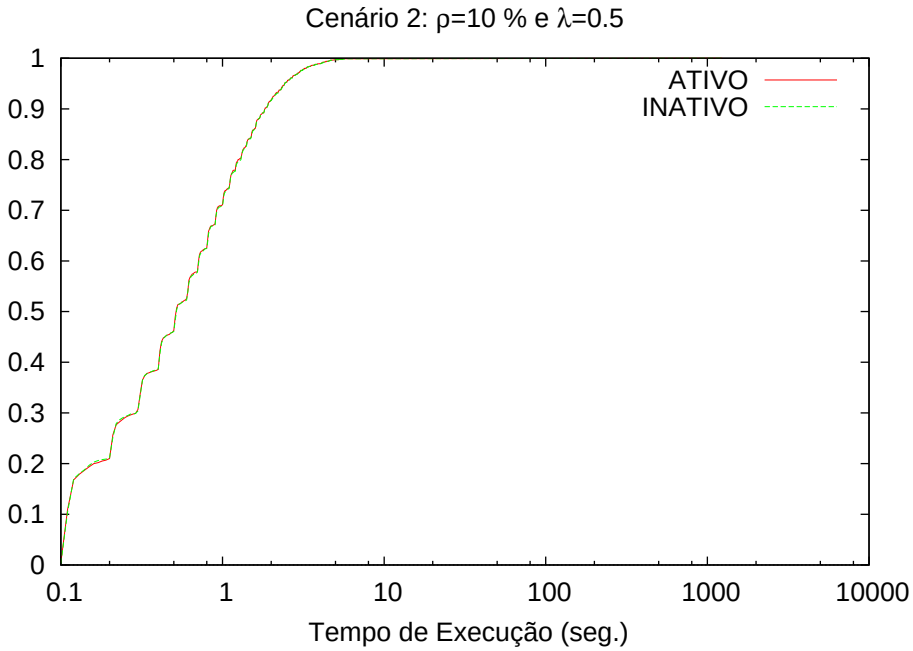

Figura 4.6: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 2*.

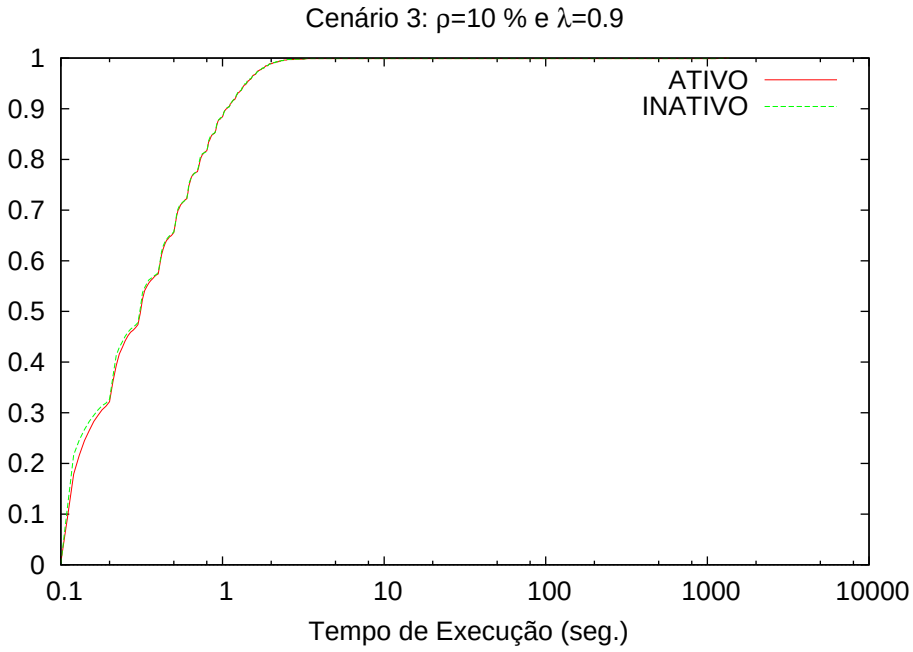

Figura 4.7: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 3*.

4.16 e 4.18).

Em relação ao desempenho dos processos remotos, observa-se que os cenários *9 e 6* apresentaram os piores desempenhos (conforme ilustra a Figura 4.4), pois esses processos além de apresentar uma alta taxa de frequência de processos locais normais ($\lambda\_local\_normal = 0{,}9$), possuem processos locais pesados com cargas grandes. A

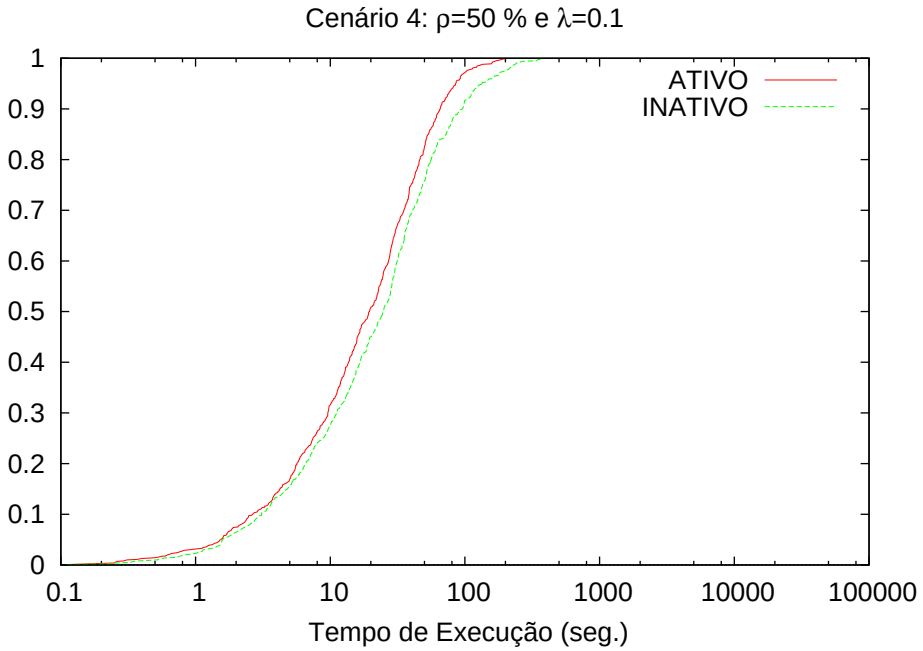

Figura 4.8: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 4*.

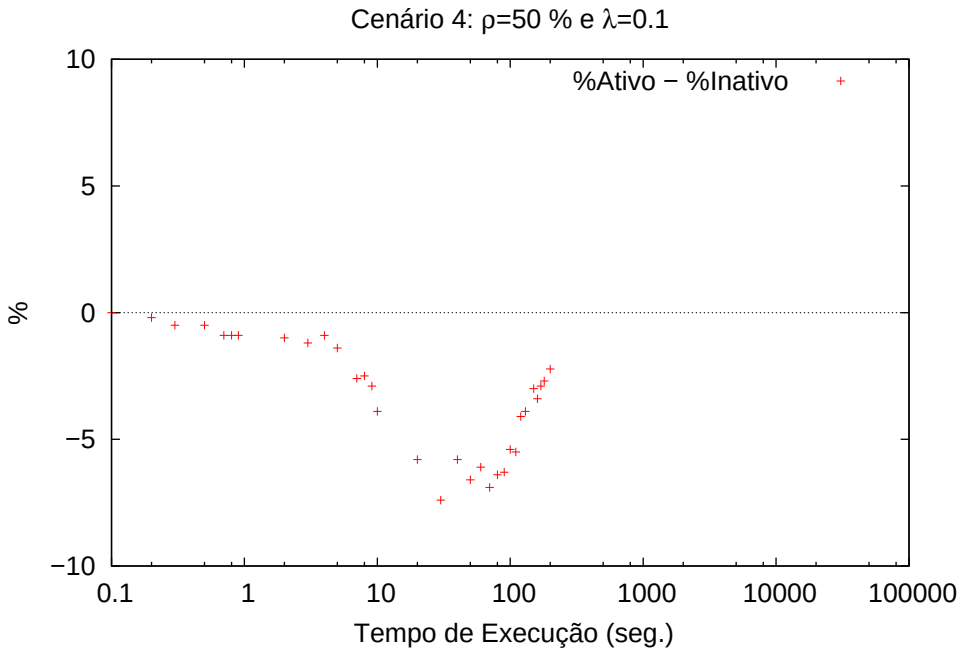

Figura 4.9: Diferença da distribuição acumulativa no *Cenário 4*.

soma desses fatores faz com que o mecanismo de restrição de recursos seja acionado constantemente e mantido ativo enquanto durarem os períodos de execução dos processos locais. Com isso, a disponibilidade de recursos para os processos oportunistas é diminuída, resultando em tempos de execução maiores quando comparados com as situações em que o mecanismo de controle não está presente. A medida que

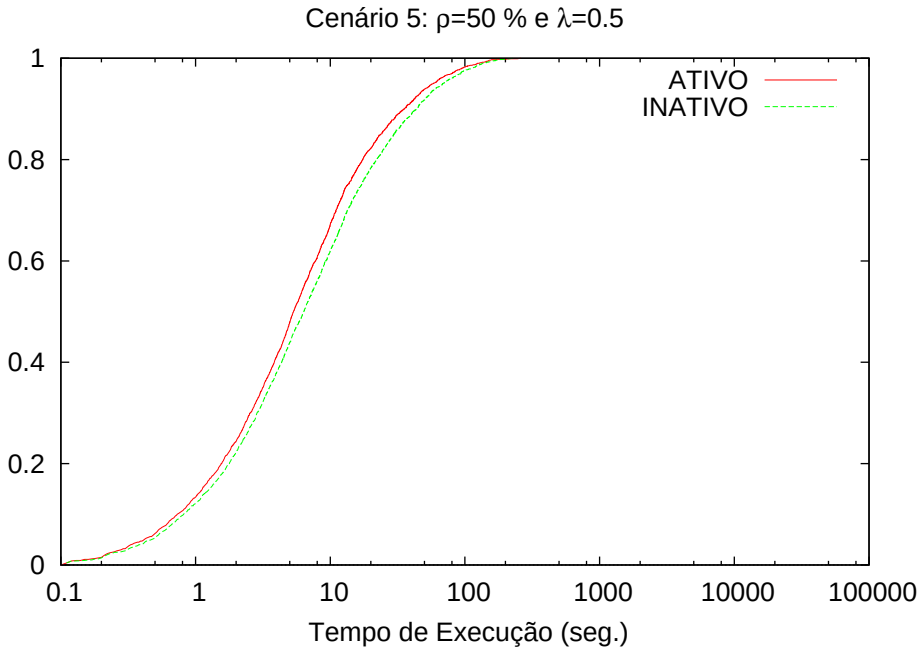

Figura 4.10: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 5*.

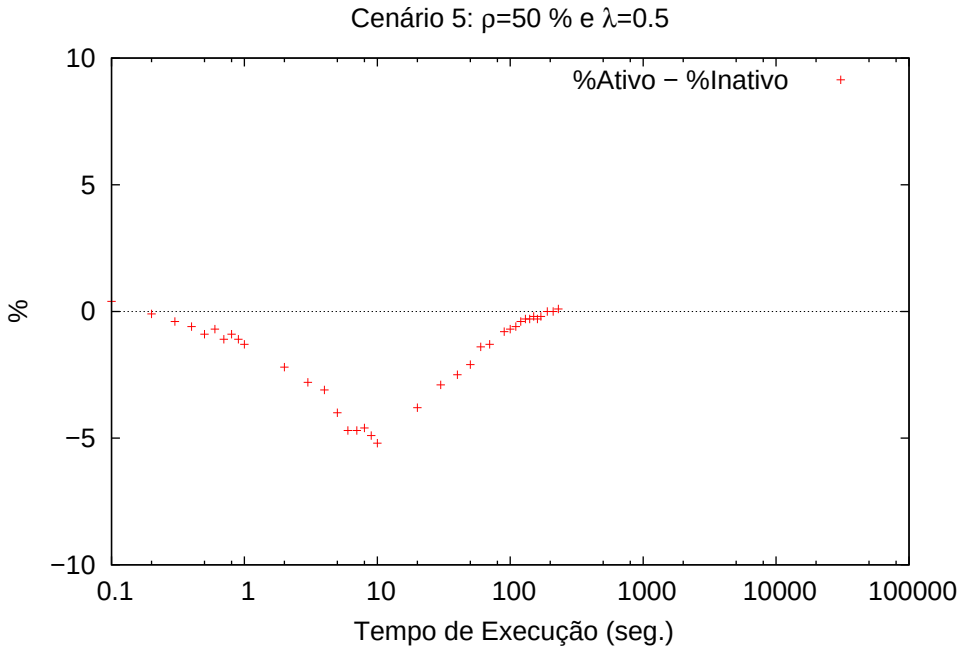

Figura 4.11: Diferença da distribuição acumulativa no *Cenário 5*.

λ_local_normal e ρ diminuem, casos dos cenários *8, 7 , 5 e 4*, a quantidade e o tamanho dos processos diminuam, fazendo com que o mecanismo seja acionado com menos frequência, permitindo que mais recursos sejam disponibilizados aos jobs.

Os casos em que as perdas foram menores (cenários com $\rho = 10\%$), os processos remotos são mais penalizados nas situações em que a taxa de chegada é maior

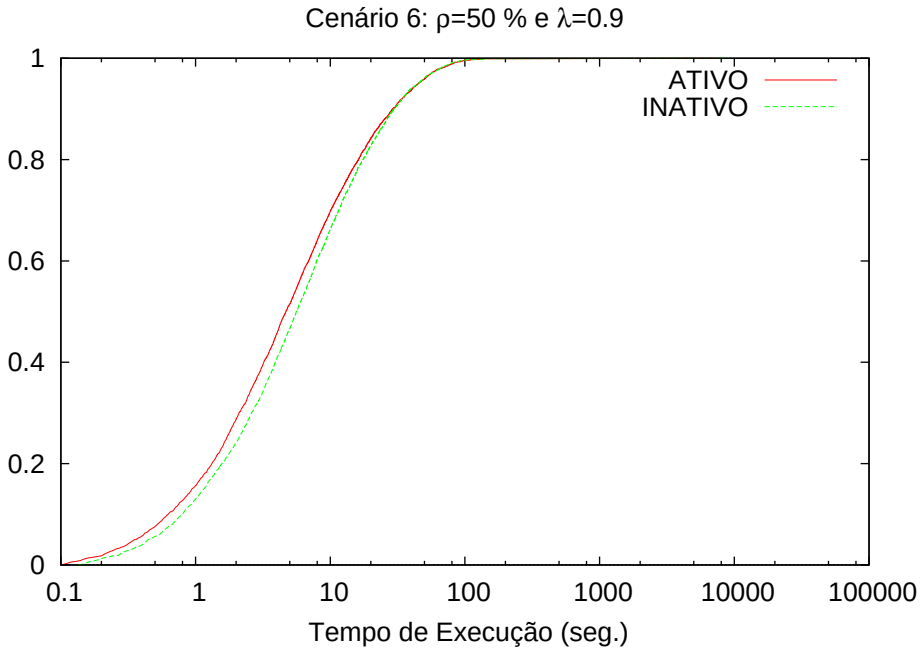

Figura 4.12: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 6*.

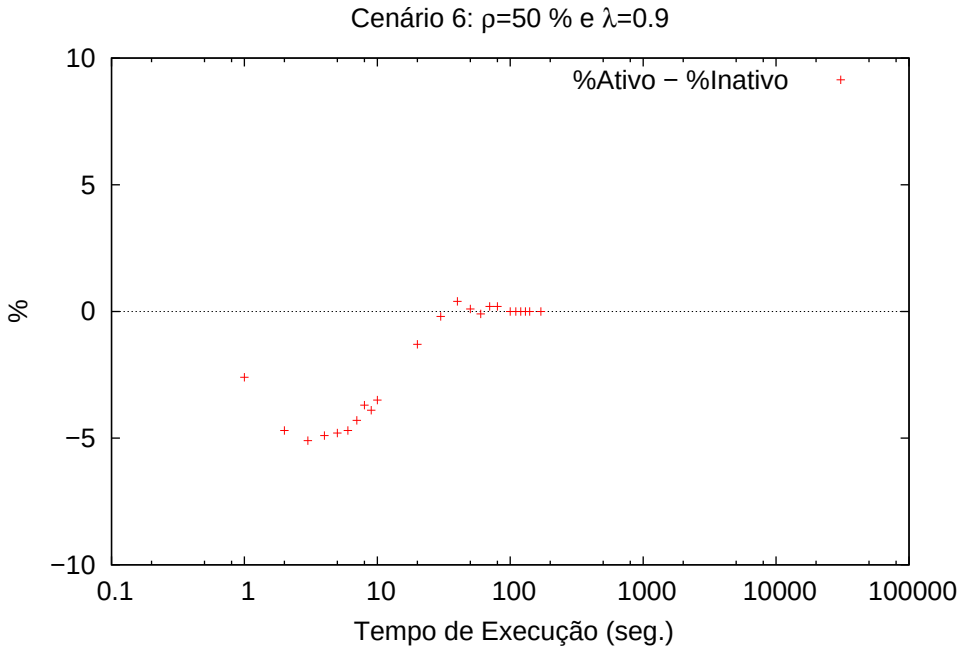

Figura 4.13: Diferença da distribuição acumulativa no *Cenário 6*.

($\lambda\_local\_normal = 0,9$) e vão sendo menos impactados a medida que chegam menos processos locais, pois acionam com pouca frequência o mecanismo, e quando o fazem, tem duração menor. Isso ocorre em virtude do tamanho dos processos ser menor que os processos dos outros cenários, principalmente, no caso dos processos locais pesados.

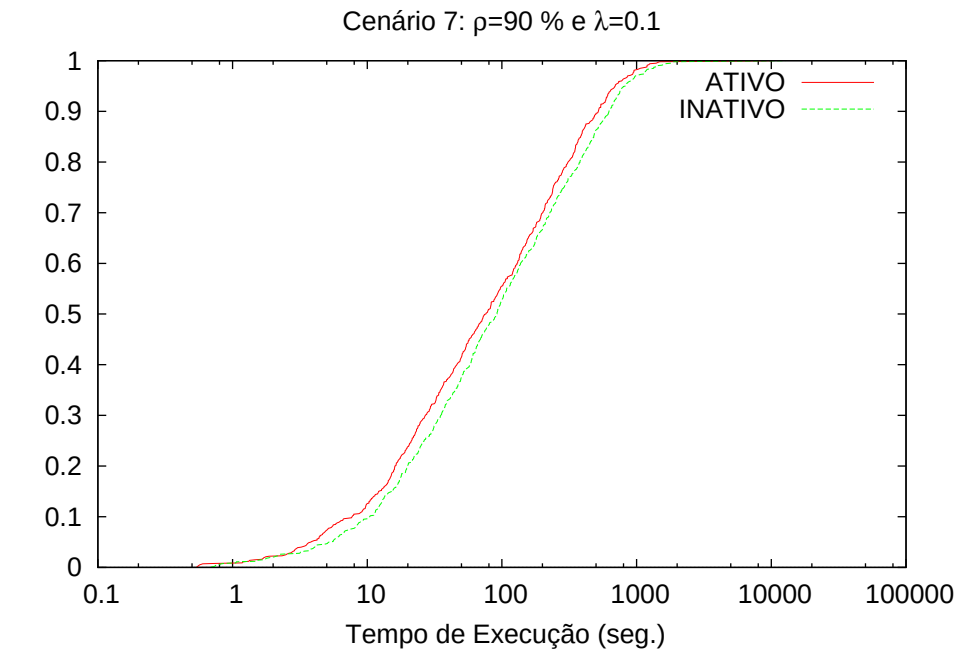

Figura 4.14: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 7*.

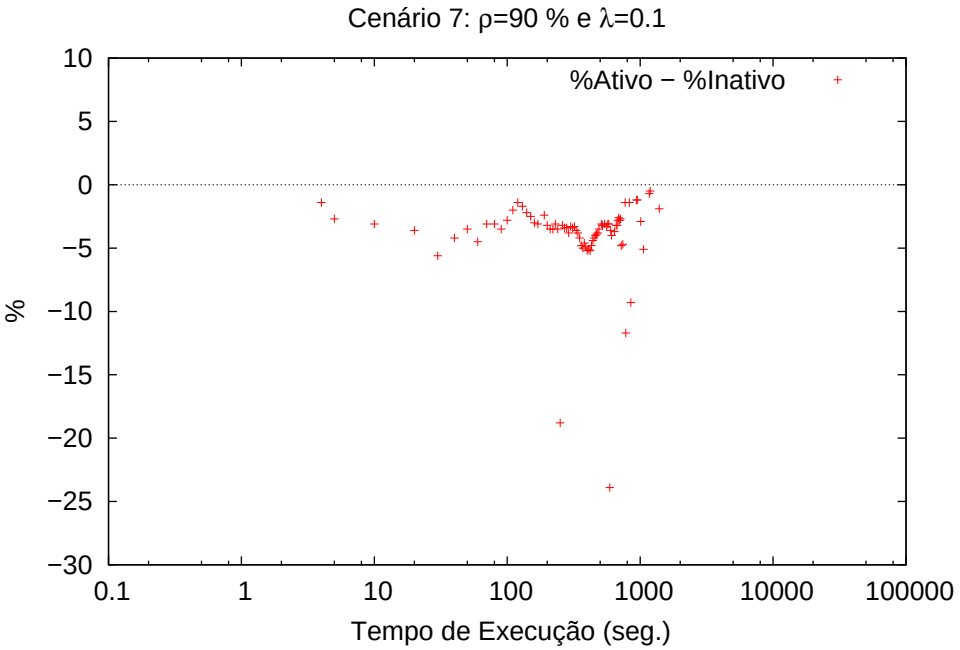

Figura 4.15: Diferença da distribuição acumulativa no *Cenário 7*.

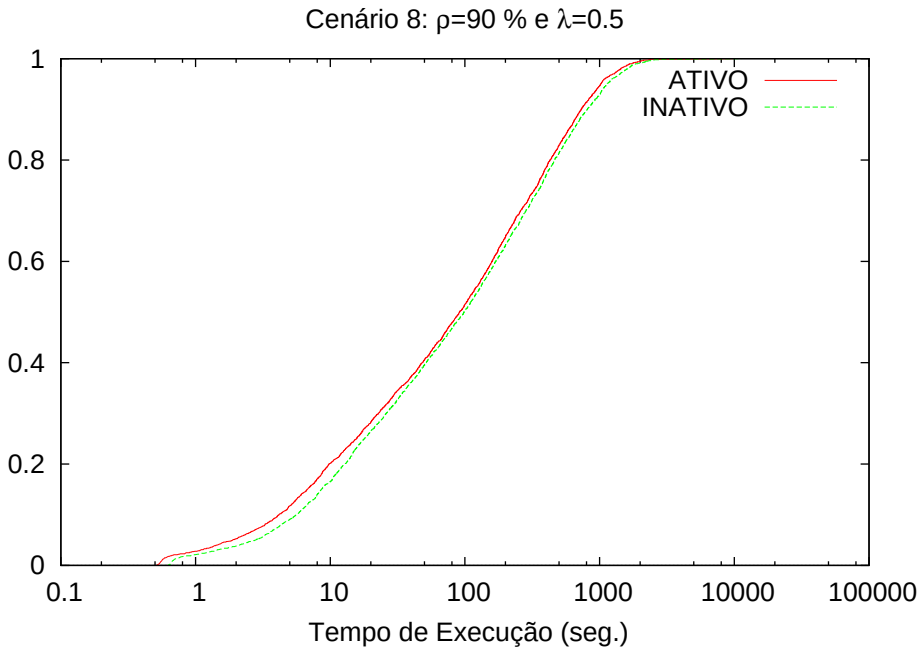

Figura 4.16: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 8*.

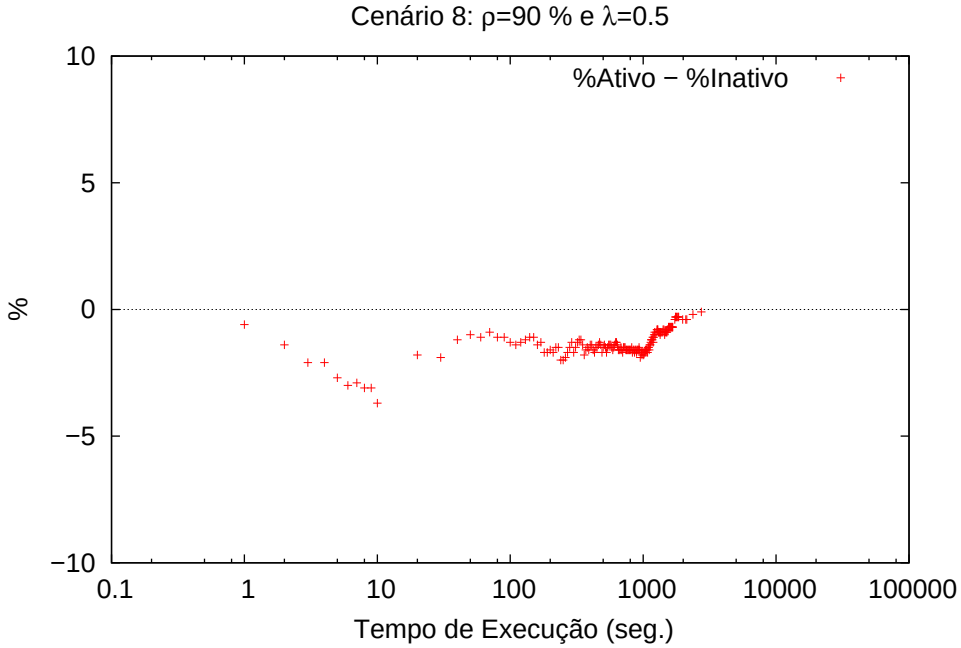

Figura 4.17: Diferença da distribuição acumulativa no *Cenário 8*.

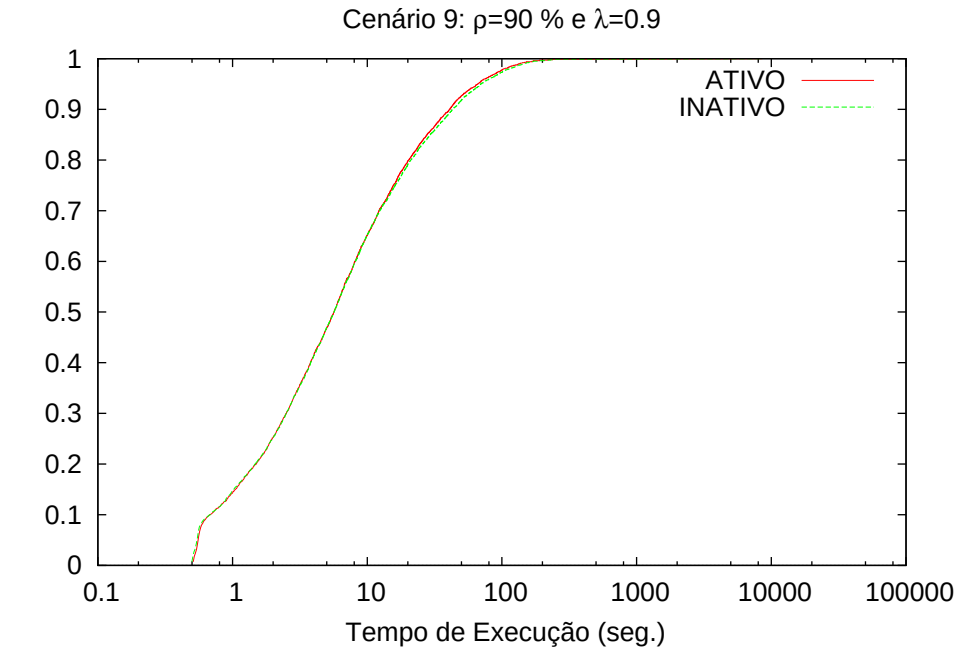

Figura 4.18: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 9*.

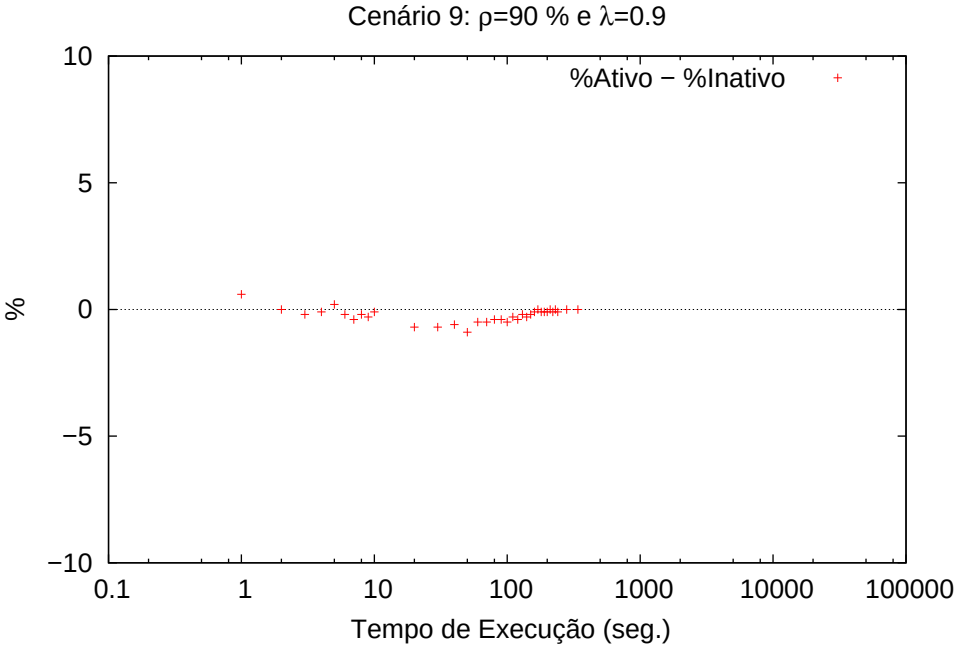

Figura 4.19: Diferença da distribuição acumulativa no *Cenário 9*.

4.1.1.2 Resultados Aplicações *CPU-Bound*: Lote - 2

Nesse lote, a principal diferença em relação ao anterior está na taxa de frequência de processos locais pesados, que é igual a taxa dos processos remotos. Dessa forma, a quantidade desse tipo de processo, comparando com os processos locais pesados dos cenários anteriores, será maior, embora com menores cargas. O desempenho do presente lote, considerando o tempo médio de execução de todos os processos com o mecanismo ativo e inativo, está apresentado na Figura 4.20. Observa-se que o comportamento do *OK* nesses cenários, para os processos locais, foi semelhante aos experimentos anteriores, os cenários que apresentaram o pior desempenho foram aqueles em que a carga de trabalho está baixa ($\rho = 10\%$ - *Cenários 1, 2 e 3*). Esses resultados são, como no caso anterior, explicados pelo fato que os processos remotos pouco impactaram os processos locais, seja: *(i)* pelo baixo número de processos locais (λ_local_normal=0,1), gerando baixa concorrência; ou, *(ii)* pelo fato que o tamanho médio dos processos locais são ainda menores (onde λ_local_normal=0,5 e 0,9), a execução destes, pouco aciona o mecanismos de restrição, podendo, inclusive, não utilizá-lo. As Figuras 4.21, 4.22 e 4.23, que mostram as respectivas distribuição acumulativas para os cenários 1, 2 e 3, expressam esse comportamento.

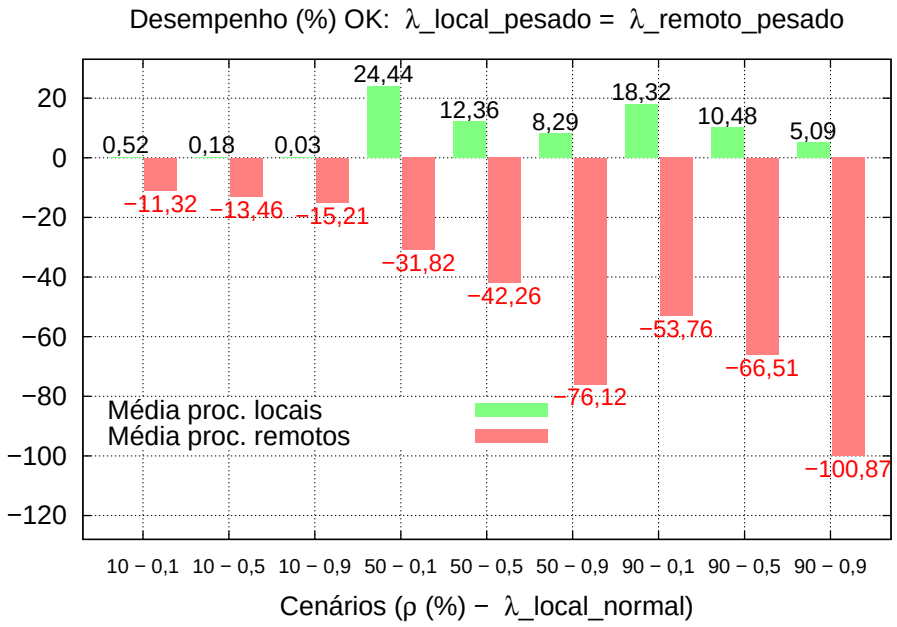

Figura 4.20: Resultado da ação do mecanismo de restrição sobre o tempo médio dos processos.

Para os demais cenários, o tempo médio de execução dos processos locais foi *menor* nos casos em que o mecanismo de controle estava atuando, principalmente, nos casos com menor valor de λ_local_normal (cenários 4 e 7), conforme observou-se nos mesmos cenários do lote anterior. Nesses casos, os processos locais normais

concorrerão menos entre si, e por serem relativamente maiores que os demais, ao concorrerem com os processos remotos, o mecanismo será acionado por mais tempo, propiciando mais recursos para os processos locais. As Figuras 4.24 e 4.30 trazem as distribuições acumulativas desses cenários.

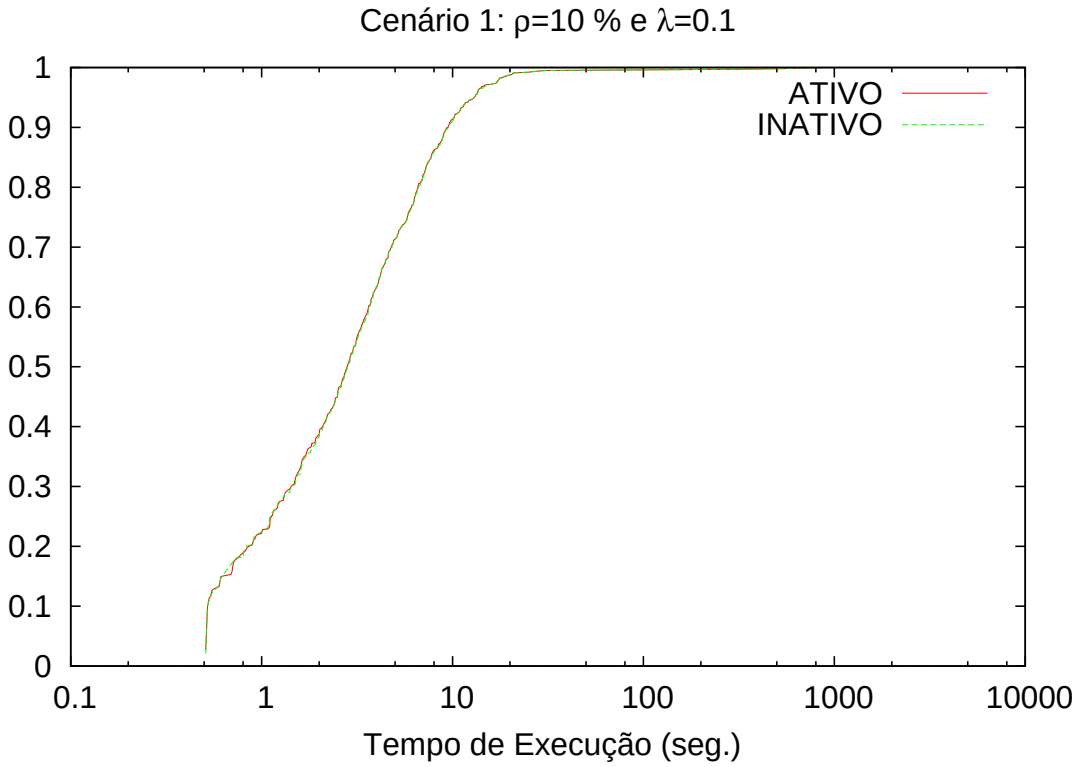

Figura 4.21: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 1* .

A Figura 4.20 mostra que, a medida que aumenta a taxa de chegada de processos (λ_local_normal) os ganhos vão diminuindo, principalmente nos casos em que ρ é igual a 50% ou 90%. Esse comportamento pode ser explicado pelo fato de que, conforme aumenta a presença de processos locais no ambiente, a concorrência entre estes pelos recursos de processamento é intensificada, fazendo com que os recursos liberados pela aplicação do mecanismo nos processos oportunistas sejam diluídos entre estes processos, resultando em menores ganhos percentuais nas comparações entre os cenários. O desempenho positivo do *OK* é, também, verificado ao se realizar a análise das distribuições acumulativas desses cenários (Figuras 4.24 a 4.34),onde é possível observar que para um dado tempo de execução, o percentual de conclusão de processos locais é maior nos casos em que o mecanismo está *ativo*. Essa conduta é mantida até a finalização do último processo, indicando que, para um mesmo grupo de processos (cenário), naquele em que houve atuação do mecanismo o tempo de execução foi menor, e por consequência foi obtido melhor desempenho.

Ao se comparar os cenários do lote-2, com os respectivos cenários do lote anterior,

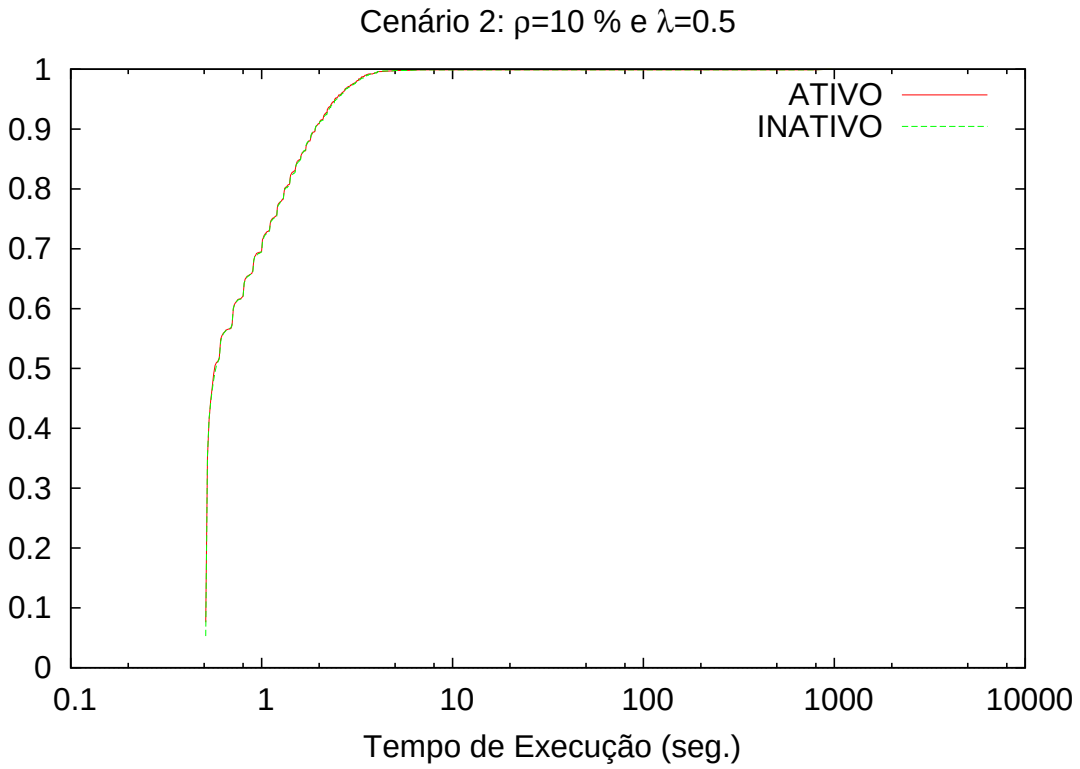

Figura 4.22: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 2*.

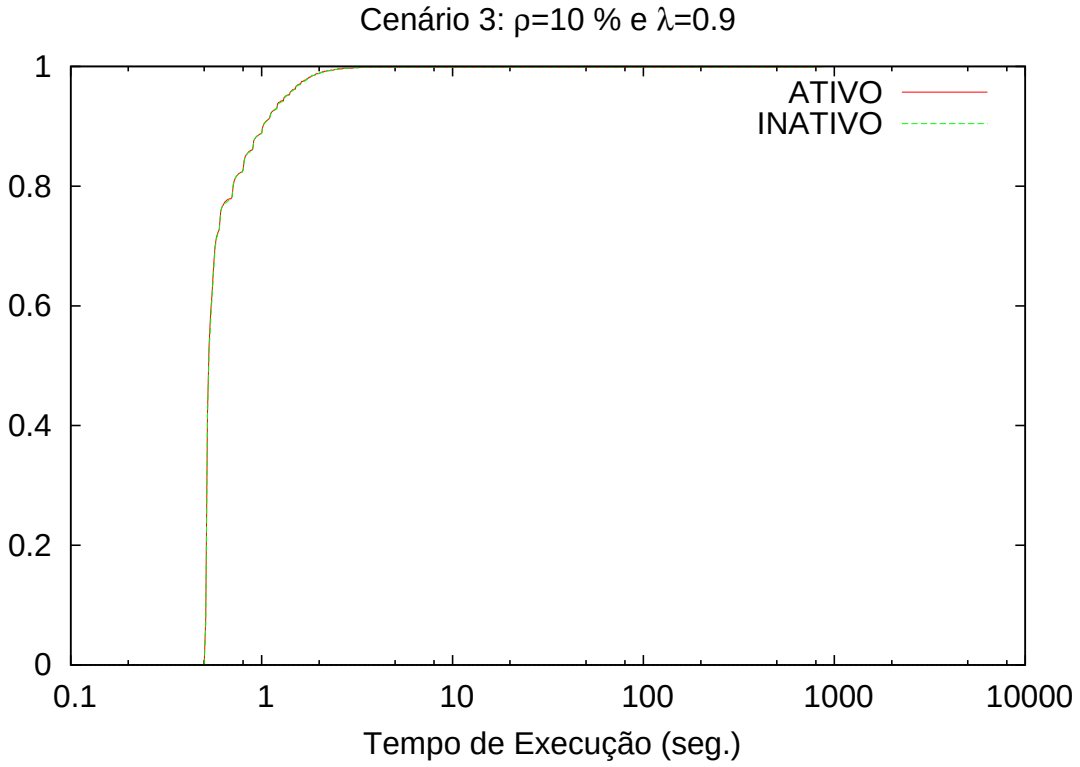

Figura 4.23: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 3* .

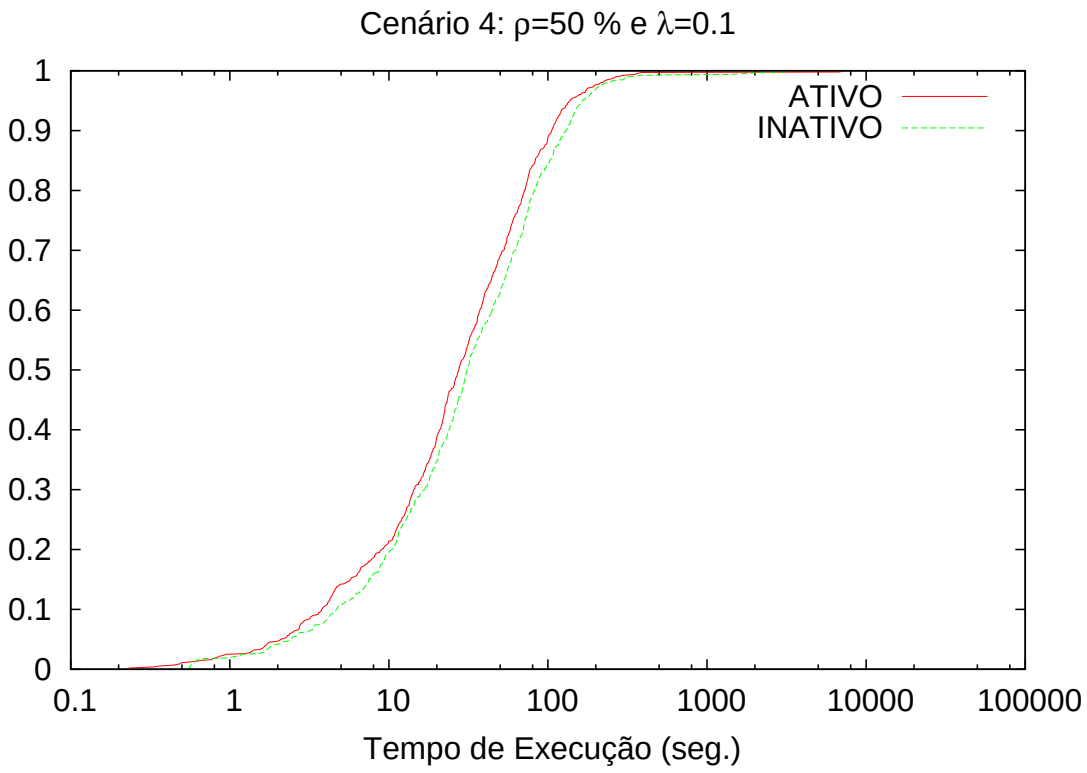

Figura 4.24: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 4*.

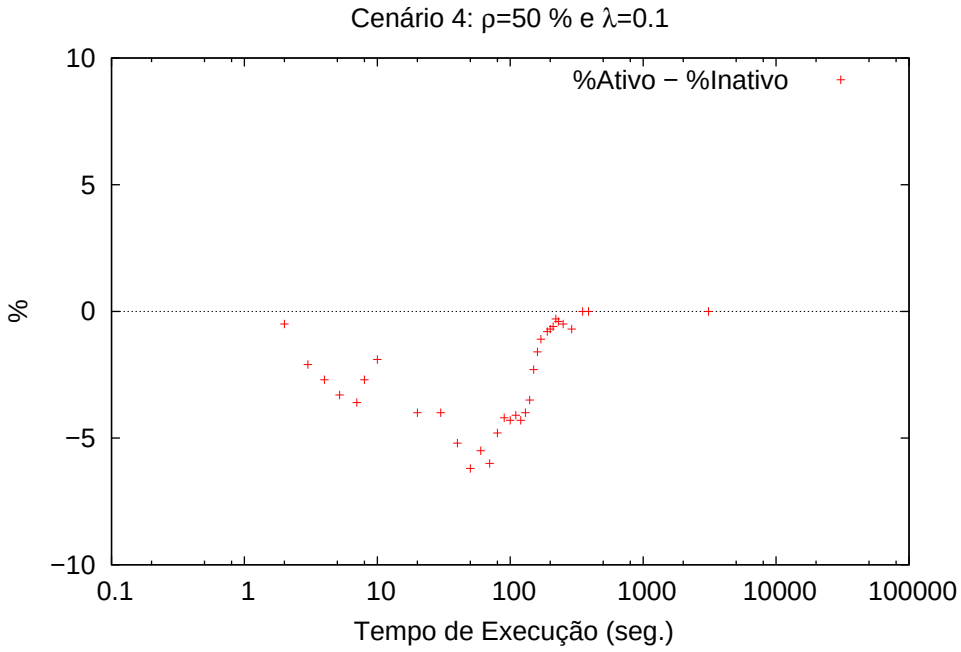

Figura 4.25: Diferença da distribuição acumulativa no *Cenário 4*.

observa-se que o desempenho daqueles foram melhores. Isso é explicado pela forma como os processos locais pesados, nos dois lotes, são afetados pelos processos remotos. Em ambos os lotes, a carga dos processos locais normais são iguais e sofrerão a

mesma interferência dos processos oportunistas, mas os processos locais pesados no segundo lote, por ter a mesma taxa de frequência dos processos remotos, possuem maior frequência e consequentemente são menores que esses processos dos cenários do lote tratado na seção anterior. Dessa forma, considerando a relação tamanho de job versus tamanho de processo local pesado, os processos pertencentes ao lote-2 serão mais afetados pelos jobs, do que os processos pesados do outro lote, que são, em média, dez vezes maiores.

Nos casos do lote-1 a soma de carga de todos os processos remotos, considerando o valor de λ_local_remoto, será menor que a carga de um processo local pesado, logo o impacto daqueles processos sobre estes, embora considerável, será diluído no tempo total de processamento dos mesmos (locais pesados). Por outro lado, os processos pesados do lote-2 possuem carga e taxa de frequência equivalentes aos processos oportunistas, ocasionando que o período proporcional de impacto gerado pelos processos remotos será, no pior dos casos, igual ao tamanho do processo local pesado, fazendo com o tempo final de execução reflita essa concorrência. Ao se ativar o mecanismo, o período de mitigação do impacto poderá ser igual ao tamanho do processo local pesado (no melhor dos casos), fazendo com que os ganhos percentuais desses cenários sejam melhores do que os outros cenários.

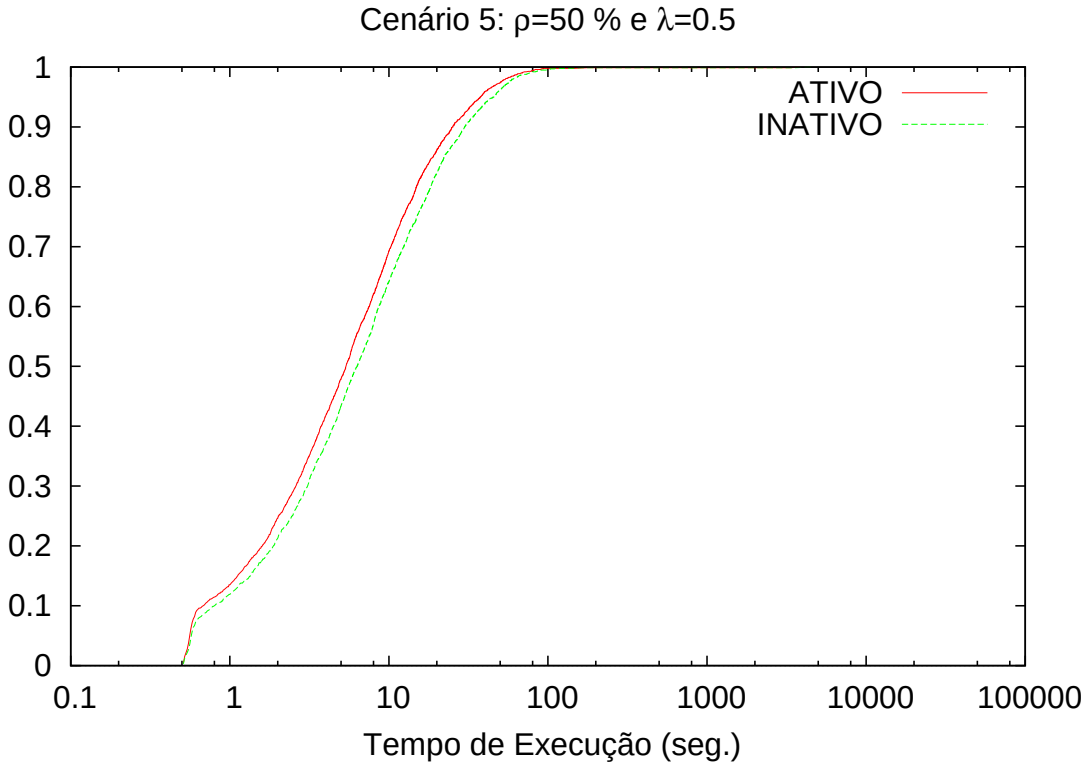

Figura 4.26: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 5*.

Os desempenhos dos cenários desse lote no que se refere aos processos oportu-

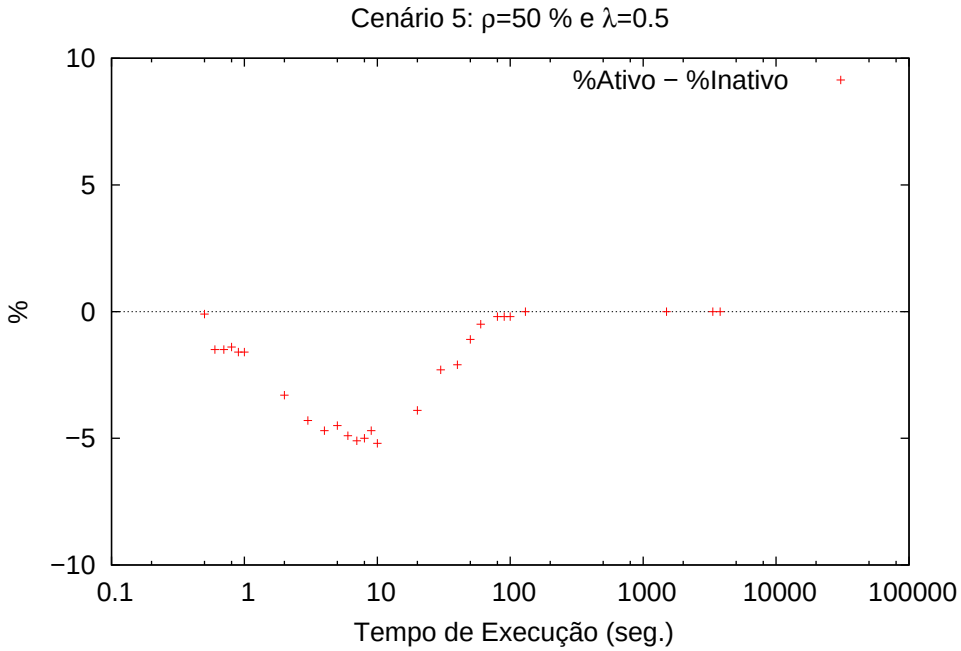

Figura 4.27: Diferença da distribuição acumulativa no *Cenário 5*.

nistas, embora tenham seguido a mesma tendência dos cenários do lote-1, apresentaram menores perdas em termos percentuais, o que é explicado pela forma como esses processos foram limitados, através do mecanismo de controle, pelos processos locais pesados. Como são menores, o tempo de atuação do mecanismo também foi menor, permitindo que os processos remotos executem com mais recursos e por consequência, seus tempos de execução foram diminuídos, considerando, os momentos em que não haviam processos locais normais. Pois se eles existissem, o mecanismo é acionado e os jobs são restringidos/penalizados normalmente.

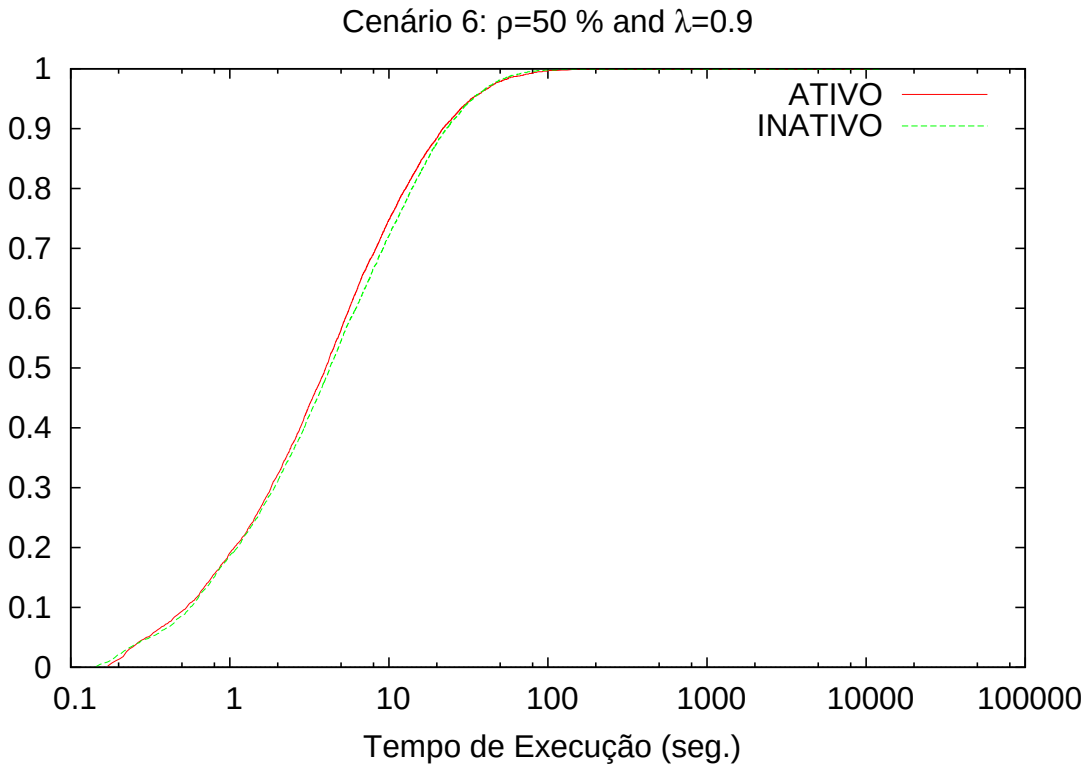

Figura 4.28: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 6*.

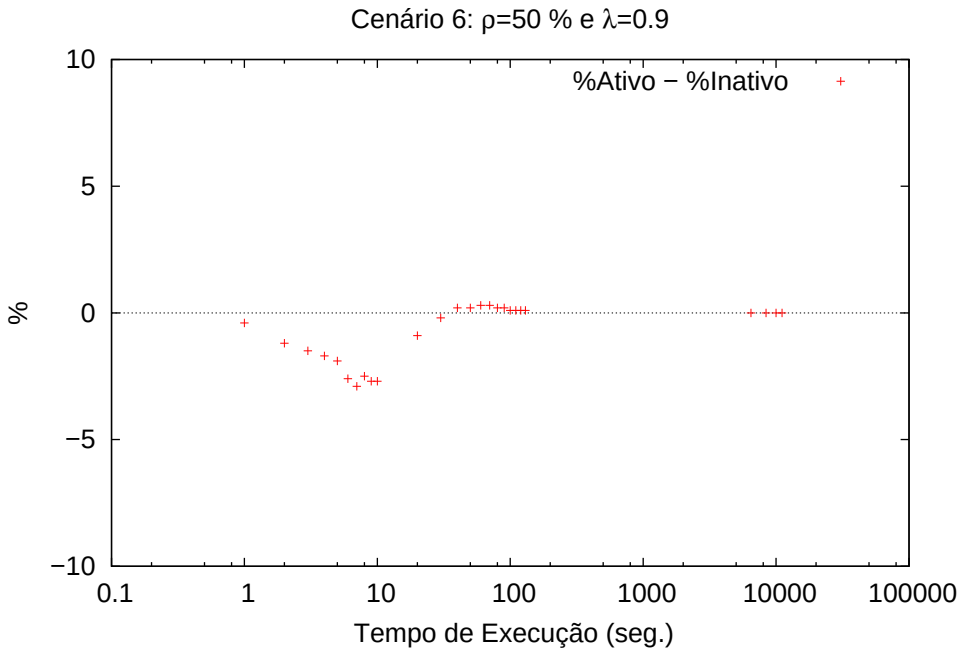

Figura 4.29: Diferença da distribuição acumulativa no *Cenário 6*.

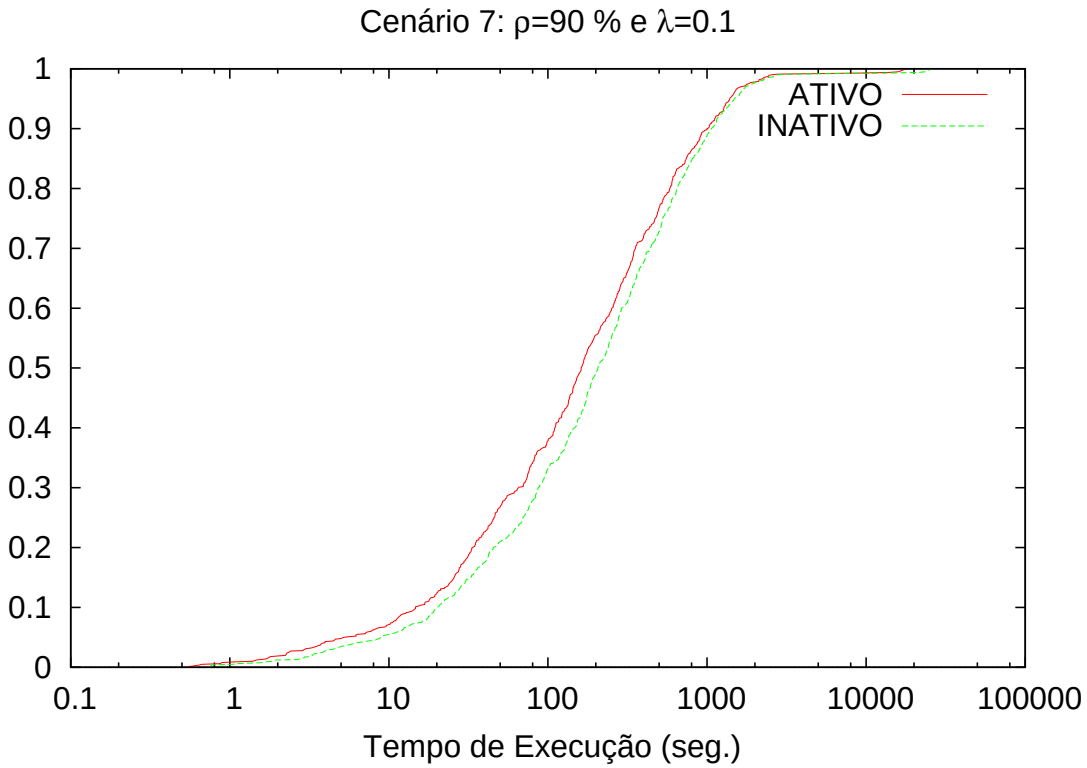

Figura 4.30: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 7*.

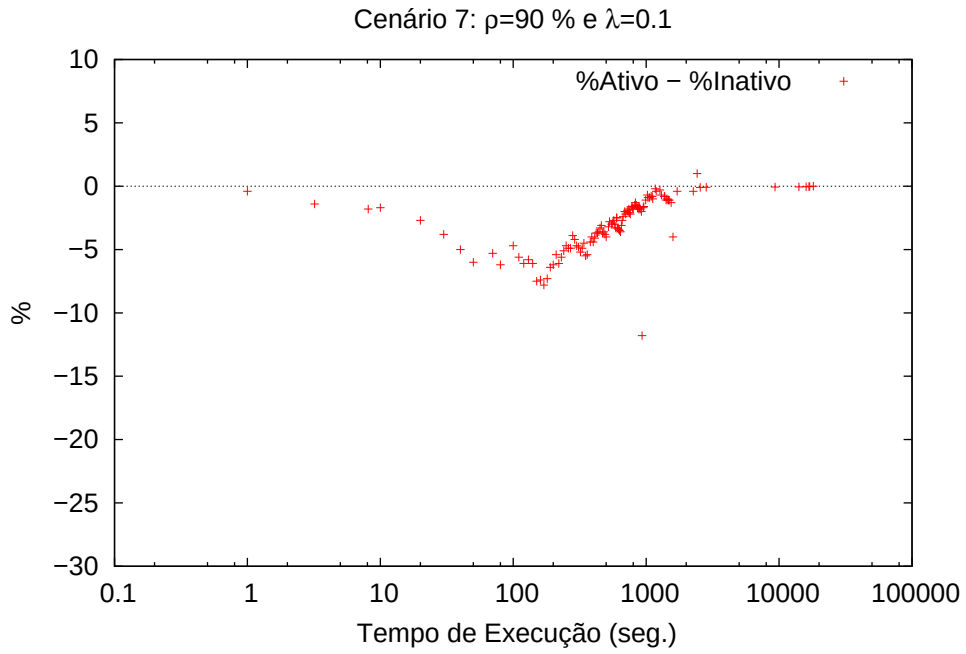

Figura 4.31: Diferença da distribuição acumulativa no *Cenário 7*.

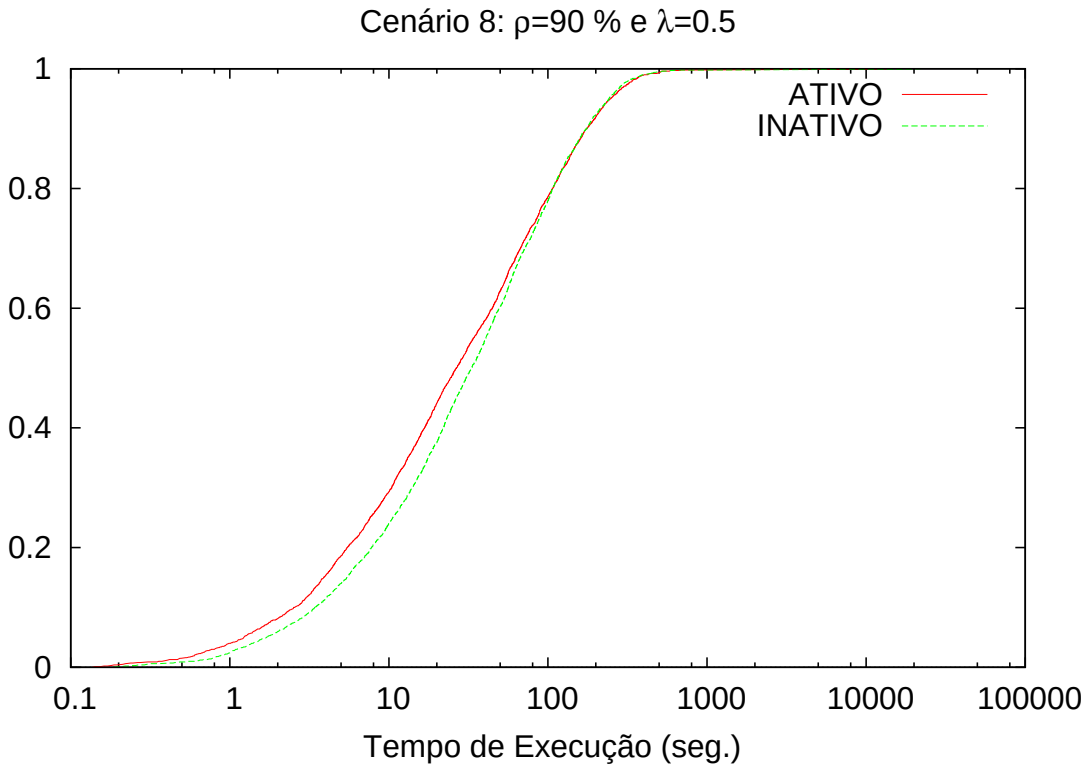

Figura 4.32: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo no *Cenário 8*.

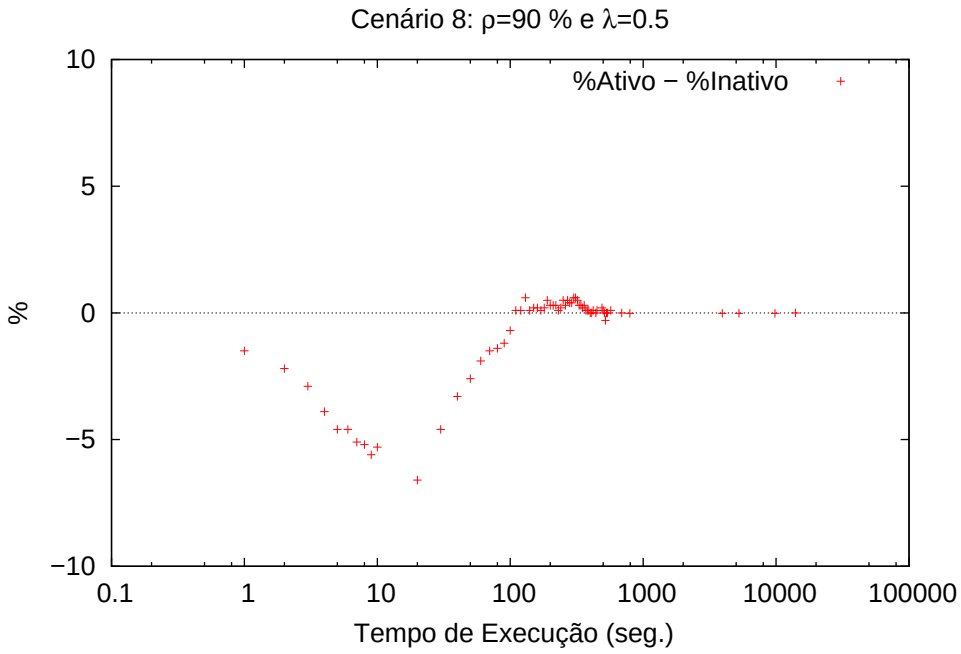

Figura 4.33: Diferença da distribuição acumulativa no *Cenário 8*.

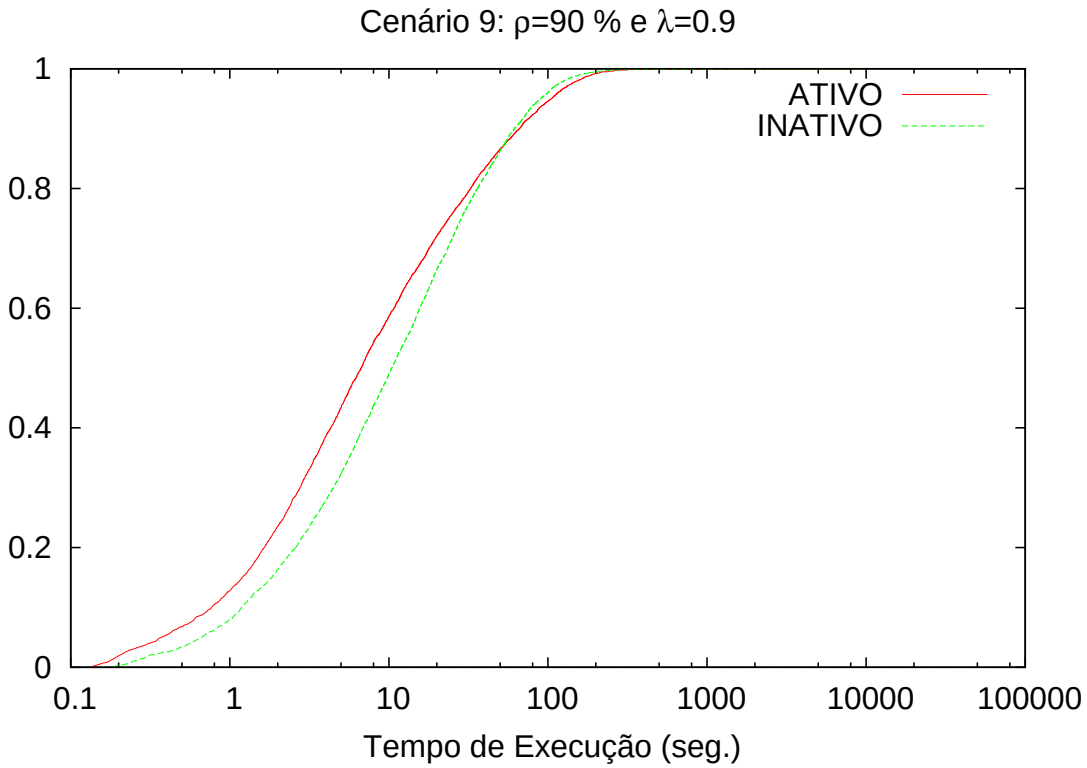

Figura 4.34: Distribuição Acumulativa dos processos locais com o mecanismo ativo e inativo nos *Cenário 9*.

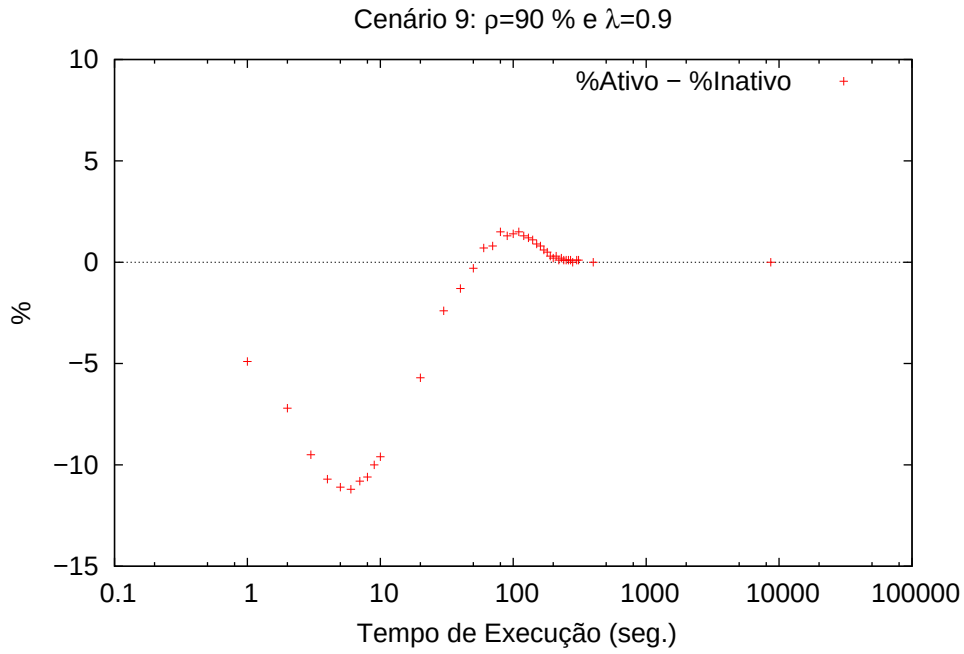

Figura 4.35: Diferença da distribuição acumulativa no *Cenário 9*.

4.1.2 Aplicações *IO-Bound*

Para essa etapa de experimentos, que trata do impacto da utilização do mecanismo sobre aplicações *io-bound*, foi utilizado, para simular processos desse tipo, o aplicativo *disktest* que faz parte da suíte LTP ([71]). A possibilidade de configurar vários parâmetros foi o motivo principal da escolha dessa ferramenta, sendo possível, entre outros:

1. Definir a quantidade percentual de operações de leitura e escrita para cada processo.

2. Determinar a quantidade de threads que acessarão o(s) arquivo(s).

3. Delimitar a quantidade de operações "seek".

4. Estabelecer o tamanho dos blocos dos arquivos.

5. Precisar o tipo de transferência de dados.

Para os experimentos, as seguintes opções foram utilizadas com a ferramenta *disktest* (as demais foram mantidas no modo *default*):

disktest -r -w -D50:50 -K1 -E32 -B8k -N100M -pR -If <arquivo>

Esse comando inicia um processo de escrita e leitura (*-r -w*) em um arquivo (*-If*) com fatias iguais de leituras e escritas (*-D50:50*). Esse processo possui uma única thread (*-K1*) e todos os dados serão verificados a procura de erros a cada 32 bytes (*-E32*). O arquivo gerado (*<arquivo>*) possui tamanho fixo equivalente a 100MB (*-N100MB*, divididos em blocos de 8KB (*-B8K*), sendo que as operações de "seek"são randômicas (*-pR*).

A avaliação da atuação do mecanismo de restrição, seguiu, em linhas gerais, a mesma filosofia dos experimentos da seção que tratou das aplicações *CPU-Bound* (seção 4.1.1), com as seguintes considerações:

1. O dado coletado em cada processo é referente ao tempo necessário para a criação e manipulação do arquivo que o comando supra-citado definiu;

2. Os processos serão gerados /classificados de acordo com a origem dos mesmos: local ou remoto;

3. O intervalo de tempo para recebimento de processos (locais e remotos) (janela de submissão *(w)*) continua sendo de duas horas (7200 segundos), não havendo limitação para término de processamento de cada processo.

As situações exploradas pelos experimentos consistem nas combinações entre a taxa de utilização de recursos de entrada e saída (*IO*) (ρ) dos processos locais (identificado como ρ_local) com a taxa dos processos remotos (denominado ρ_remoto).

O valor de ρ para esses dois tipos de processos são: *10%, 50% e 90%*, indicando, respectivamente, pequena, média e alta utilização de recursos. Dessa forma, o total de situações (combinações) a serem examinadas é 9(nove). São elas:

Situação 1 : $\rho\_local = 10\%$ e $\rho\_remoto = 10\%$

Situação 2 : $\rho\_local = 10\%$ e $\rho\_remoto = 50\%$

Situação 3 : $\rho\_local = 10\%$ e $\rho\_remoto = 90\%$

Situação 4 : $\rho\_local = 50\%$ e $\rho\_remoto = 10\%$

Situação 5 : $\rho\_local = 50\%$ e $\rho\_remoto = 50\%$

Situação 6 : $\rho\_local = 50\%$ e $\rho\_remoto = 90\%$

Situação 7 : $\rho\_local = 90\%$ e $\rho\_remoto = 10\%$

Situação 8 : $\rho\_local = 90\%$ e $\rho\_remoto = 50\%$

Situação 9 : $\rho\_local = 90\%$ e $\rho\_remoto = 90\%$

De maneira semelhante aos cenários *CPU-Bound*, o ponto de partida para cada situação é a geração de duas listas (uma para cada tipo) contendo a carga e o $momento(m)$ em que cada processo deve entrar em execução. Como os experimentos utilizam os recursos de entrada e saída (IO), os valores λ e μ utilizados para criação dessas listas, são obtidos em função da capacidade da controladora de discos do nó executor, que é igual a 7,69 MB/s.

Sendo assim, há o servidor (controladora) que entrega dados a uma certa taxa μ dada por 7,69 MB/s. A taxa λ de chegada de pedidos para esse servidor depende de quantos processos chegam por segundo e da carga de IO que cada um traz. Mantendo a carga l constante, é possível fazer o cálculo da taxa de chegada de processos por segundo. Como o ρ agora se refere à controladora, ao realizar $\lambda = \rho \times \mu$, obtém-se um número expresso em MB/s. A quantidade de processos que chegarão em média por segundo, pode ser calculada através da divisão de λ por l. O inverso do resultado da divisão será o tempo médio entre chegadas e este, como no caso dos ensaios *cpu-bound*, será usado para gerar a lista de chegada de processos.

Para exemplificar esses cálculos, para a situação em que $\rho = 0{,}5$. Calcula-se $\lambda = 0,5 \times 7,69 = 3{,}845$ MB/s. Para a controladora ter essa média de chegadas com cada processo tendo uma carga de 100 MB, é necessário ter 3,845 / 100 = 0,03845 processos por segundo, ou seja, os processos precisarão chegar com um intervalo médio de tempo entre eles dado por 1 / 0,03845, que é igual a 26 segundos. Portanto, os instantes de execução *(t)*, que compõem o $momento(m)$, serão calculados com da

Fórmula 4.2, onde o valor de λ será aquele obtido anteriormente (no caso do exemplo, o valor de λ é 0,03845).

Depois que as duas listas são criadas, ocorre a concatenação e ordenação (por m) das mesmas, resultando na *lista de entrada* da situação. Esta, por sua vez, é executada pelo nó duas vezes (uma vez com o mecanismo *ativo* e outra com o mecanismo *inativo*), coletando-se o tempo de criação e manipulação do arquivo respectivo a cada cada processo.

Cada situação, com listas de entradas diferentes, foi executada por quatro vezes[7], e os tempos coletados relativos a cada situação foram concatenados em um único resultado. A seguir os resultados obtidos serão analisados.

Os resultados desses experimentos (médias dos tempos de execução dos processos) estão ilustrados na Figura 4.36. É importante ressaltar que alguns desses experimentos ($\rho\_local = 50\%$ e $\rho\_remoto = 90\%$, $\rho\_local = 90\%$ e $\rho\_remoto = 50\%$, $\rho\_local = 90\%$ e $\rho\_remoto = 90\%$), ao ultrapassarem a capacidade nominal da controladora, causaram algumas distorções nos resultados apurados, por isso estes resultados não estão apresentados. Para os demais casos observa-se que os ganhos foram pequenos, condição já esperada, uma vez que o mecanismo só é acionado quando há demandas por recursos de processamento, o que não é caso desses experimentos. Nas situações em que houve grande concentração de processos remotos e o(s) processador(es) tenha(m) sido utilizado(s), o mecanismo foi posto em execução, mesmo que em curtos períodos de tempo, como é o caso da coluna 10%, 90% da Figura 4.36. Nos casos em que a quantidade de processos oportunistas foi baixo (colunas 10%, 10% , 50%, 10% e 90%, 10%), o mecanismo foi pouco acionado, resultando em pequenos ganhos para os processos locais.

[7] As exceções ficam por conta das situações $\rho\_local = 50\%$ e $\rho\_remoto = 90\%$, $\rho\_local = 90\%$ e $\rho\_remoto = 50\%$, $\rho\_local = 90\%$ e $\rho\_remoto = 90\%$ que apresentaram problemas durante as simulações e foram rodadas três vezes

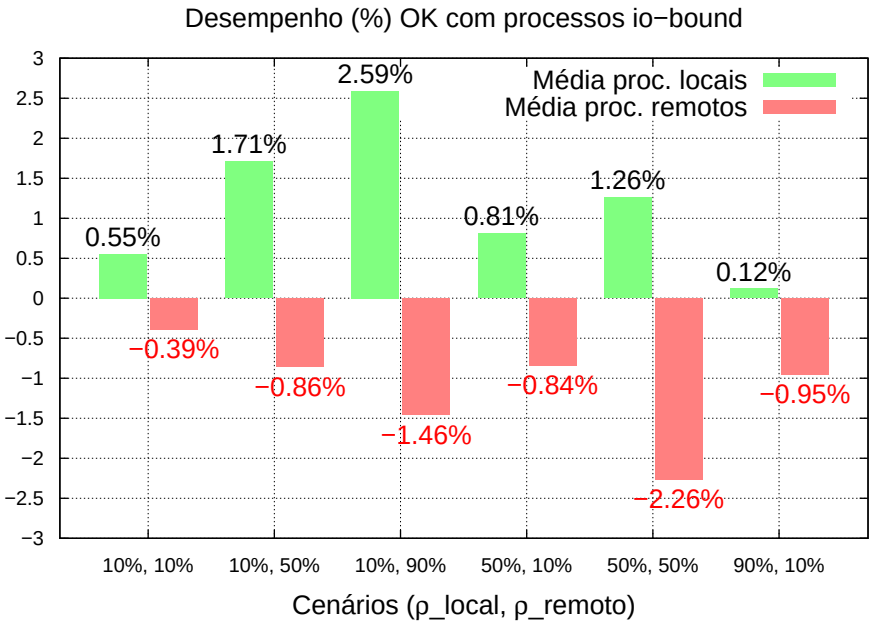

Figura 4.36: Resultado da ação do mecanismo de restrição sobre o tempo médio dos processos io-bound.

4.1.3 Aplicação *CPU-Bound Real*

Atualmente, este mecanismo está instalado e configurado em um grid para processamento de aplicações G & G (*Geology and Geophysics*), que é composto por estações de usuários (nós de processamento) da companhia. O mecanismo de controle permite que os processos provenientes do grid (processos oportunistas), sejam executados nas estações dos usuários concorrentemente com os processos destes, de tal forma que o impacto causado pelos jobs aos processos locais seja minimizado nas condições em que há disputa por recursos, e a medida que os recursos vão sendo liberados, estes serão absorvidos pelos processos remotos.

Para apresentar os resultados da atuação do mecanismo nesse ambiente de produção, foram selecionadas as situações mais comuns que acontecem nesse ambiente quando há aplicações G&G locais e remotas [8] competindo por recursos de processamento. Os processos foram divididos, considerando o tempo médio de processamento dessas aplicações, em dois grupos: processos *normais* e processos *pesados*. Sendo que os casos experimentados foram as configurações, com exceção do primeiro, mais comuns que são encontradas nesse grid, conforme lista abaixo (→ indica a sequencia de entrada em execução):

Caso 1 : Local-Pesado_1 e Remoto-Pesado_1, inciando ao mesmo tempo.

[8] A aplicação utilizada nos experimentos é um simulador de reservatórios de petróleo denominado IMEX [72], que tem por característica, após o carregamento dos dados, utilizar intensamente o(s) processador(es) do nó executor e conforme a simulação/processamento ocorre, os dados são escritos em arquivos de saída.

Caso 2 : Local-normal_1 → Remoto-Pesado_1 → Local-normal_2 → Local-normal_3;

Caso 3 : Remoto-Pesado_1 → Local-normal_1 → Local-normal_2 → Local-normal_3;

Caso 4 : Remoto-normal_1 → Local-normal_1 → Local-normal_2 → Remoto-normal_2 → Local-normal_3 ;

No primeiro caso, as Figuras 4.37 e 4.38, mostram o comportamento do mecanismo na situação em que um processo local pesado concorre com um processo remoto também pesado, ambos iniciados simultaneamente. Nesse caso, o tempo de execução do processo local com o mecanismo ativo (Figura 4.38) foi de *21.864,08″*, enquanto o tempo de execução desse processo com o mecanismo inativo (Figura 4.37) foi de *41.160,58″*, representando um desempenho 46,88% melhor com a adoção do mecanismo de controle. Esse desempenho é resultado da priorização que foi dada ao processo local em relação ao processo remoto, como é possível observar na Figura 4.38.

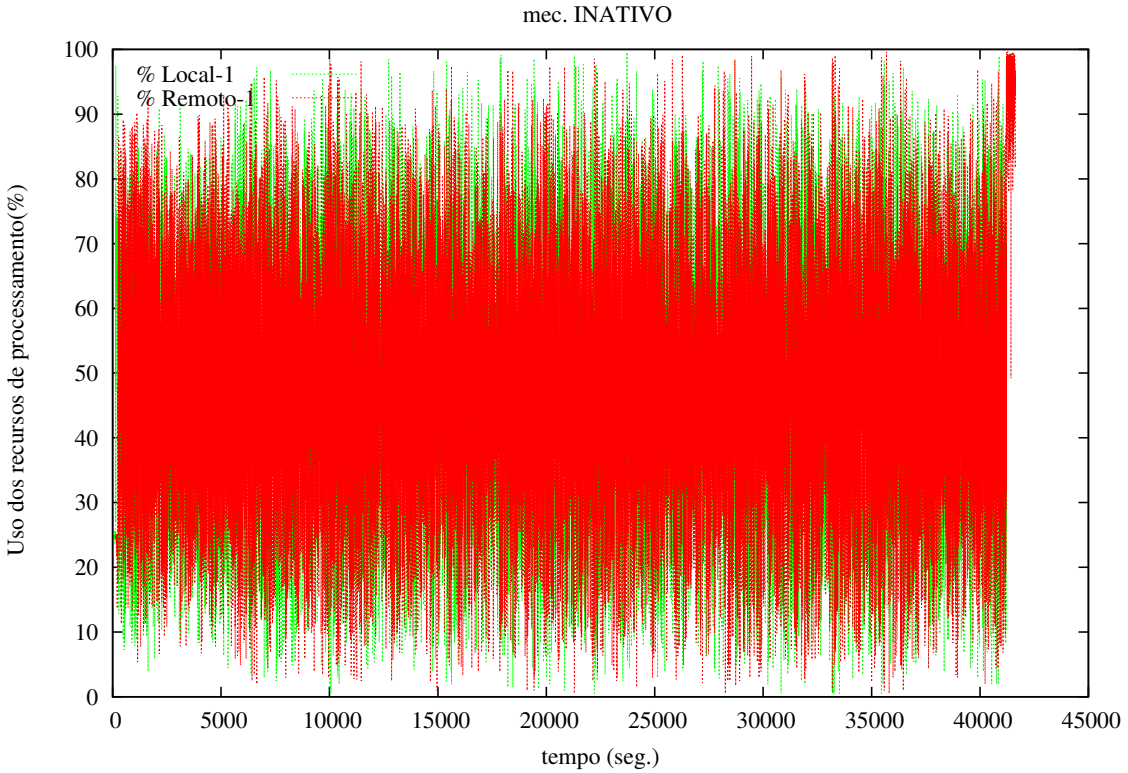

Figura 4.37: Caso 1 - Competição por recursos entre processo local pesado e processo remoto pesado com o mecanismo INATIVO

Quando o processo local terminou a execução, todos os recursos foram alocados exclusivamente para o processo remoto, o que além de eliminar a necessidade de trocas de contexto entre processo, diminuiu a necessidade de intervenção do mecanismo

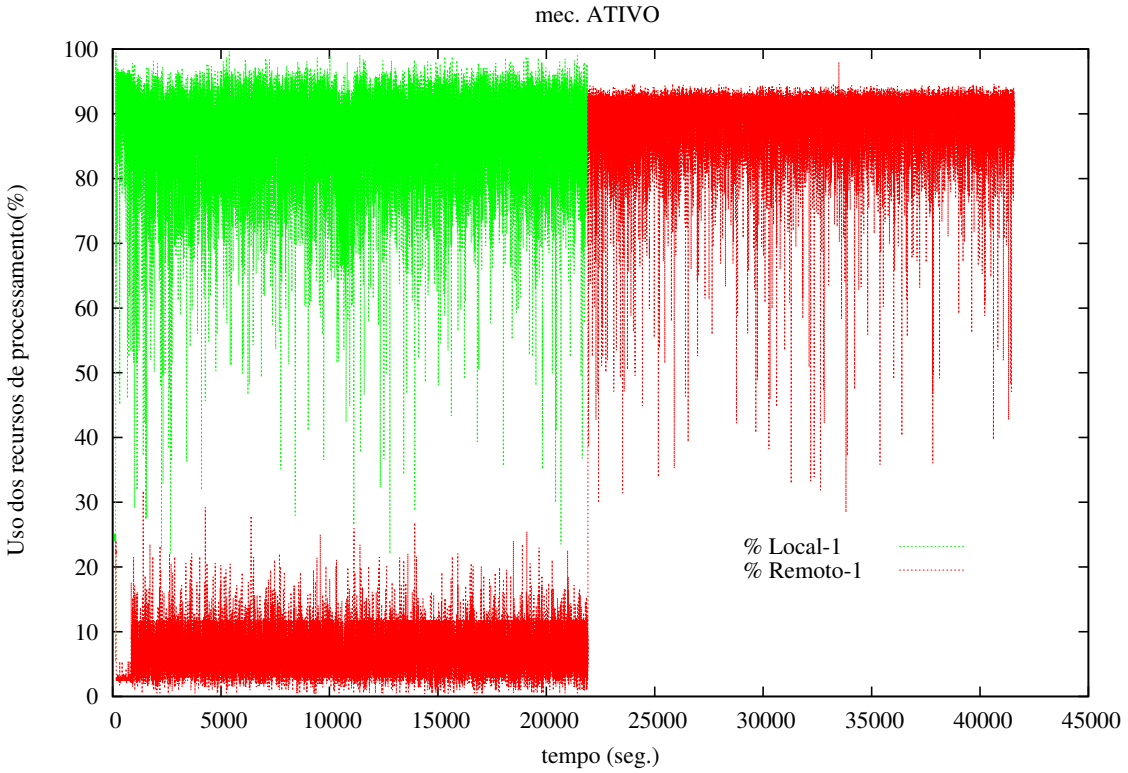

Figura 4.38: Caso 1 - Competição por recursos entre processo local pesado e processo remoto pesado com o mecanismo ATIVO

de controle sobre os processos. Com isso o processo remoto com o mecanismo ativo demandou *42.422,02″*, valor 2,29% superior ao tempo obtido com o mecanismo inativo (*41.471,30″*).

O tempo de execução desse processo local, sem a concorrência de qualquer outro processo oportunista e com o mecanismo de controle desligado, foi de *19.763,97″*, esse desempenho é considerado ideal para o usuário, pois todos os recursos do nó estarão a disposição para o seu processo. A Figura 4.39 traz a alocação dos recursos do início até o término da execução desse processo, nota-se que o consumo manteve-se próximo do máximo da capacidade do nó. Ao ser introduzido um job a distribuição é aquela apresentada na Figura 4.37, resultando em um tempo maior de execução do processo local, que é prioritário, ou seja, o processo remoto impactou o usuário desse nó de processamento. Analisando as Figuras 4.38 e 4.39, mesmo possuindo desempenho inferior, a atuação do mecanismo propiciou que a alocação de recursos ao processo local se assemelhasse com a situação em que o processo executa de forma isolada, diminuindo o impacto causado pelo processo oportunista e possibilitando a utilização desse nó de processamento como parte integrante do grid.

O próximo experimento (*Caso 2*) mostra o impacto causado aos processos locais causado pelo processo oportunista pesado, nesse caso, o processo Local-1 inicia sua execução e após um período de tempo o processo Remoto-1 é iniciado, da mesma

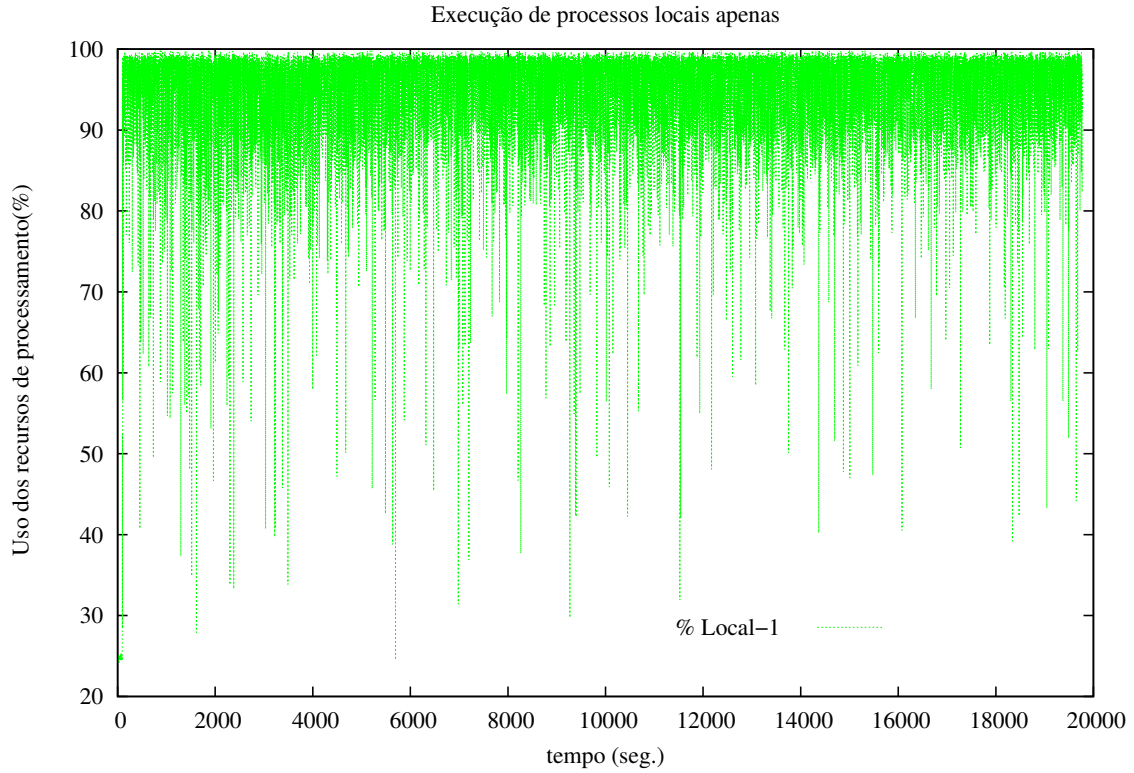

Figura 4.39: Caso 1 - Processo local executando de forma isolada e sem mecanismo de controle.

forma, os outros dois processos (Local-2 e Local-3) são colocados em execução em momentos diferentes. A Figura 4.40 mostra a distribuição dos recursos aos processos desse caso *sem* a interferência do mecanismo de controle. É possível verificar que o processo Local-1 executou sem concorrência até a chegada do processo Remoto-1 (indicado por (a) na Figura 4.40), consequentemente, os recursos que estavam totalmente dedicados a execução do processo local, tiveram que ser compartilhados com o processo oportunista. Com essa disputa, esse processo local não conseguiu ser executado antes que o próximo processo (Local-2) fosse colocado em execução (indicado por (b) na Figura 4.40), colaborando para que o tempo de execução de ambos aumentasse. O último processo (Local-3) ao ser lançado (indicado por (c) na Figura 4.40), embora não tenha sido impactado pelos processos locais anteriores, foi prejudicado pela concorrência vinda do job, o que resultou em um tempo maior de execução. Devido ao consumo de recursos por parte do processo oportunista, os tempos de execução (em segundos) para os processos Local-1, Local-2, Local-3 foram, respectivamente, *1.282,96″*, *1.718,83″* e *1.802,96″*. Enquanto o processo oportunista executou em *21.830,44″*.

Quando os mesmos processos foram novamente executados *com* o mecanismo ativado (Figura 4.41), a quantidade de recursos alocados para os processos locais, nos períodos em que estavam concorrendo com o job remoto, devido a atuação

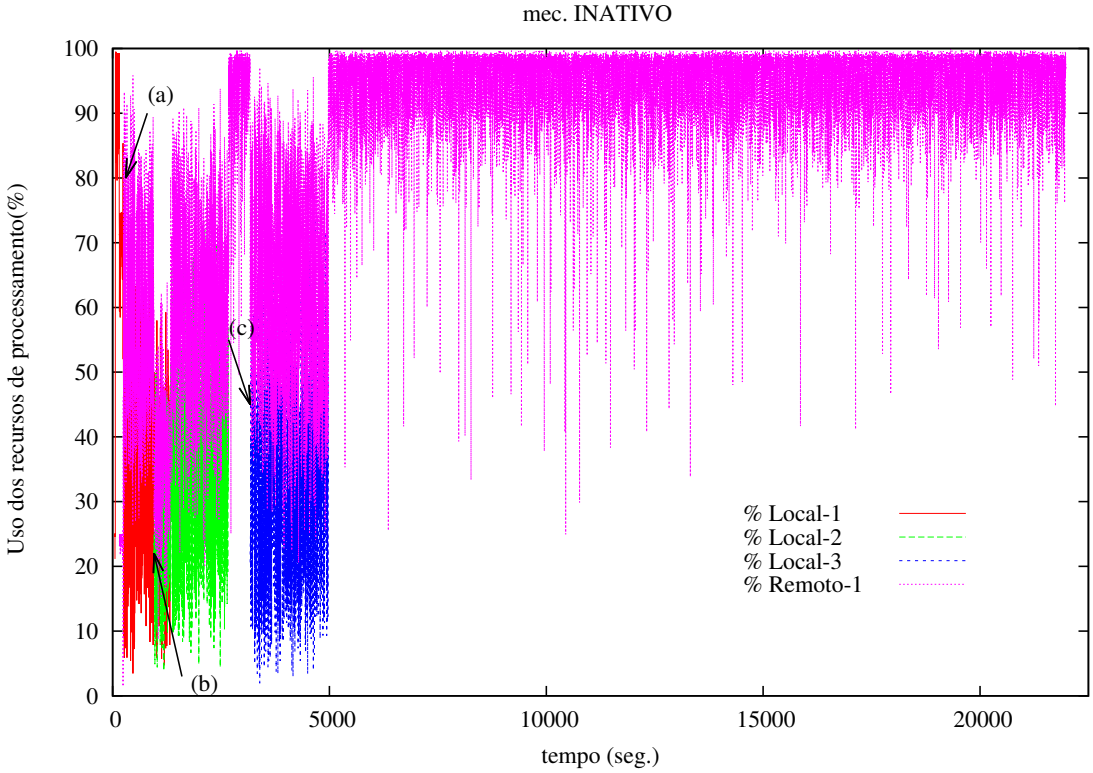

Figura 4.40: Caso 2 - Competição por recursos entre 1 processo local pesado e 3 processos remotos normais com o mecanismo INATIVO

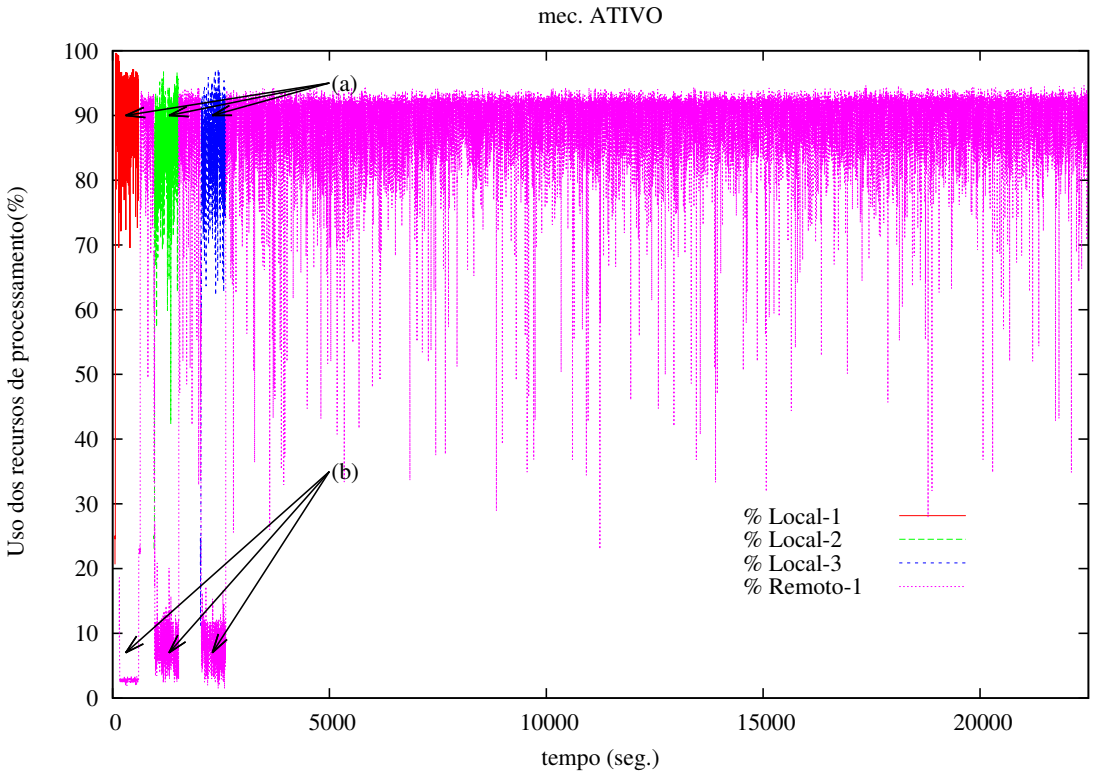

Figura 4.41: Caso 2 - Competição por recursos entre 1 processo local pesado e 3 processos remotos normais com o mecanismo ATIVO

mecanismo, aproximou-se do máximo permitido pelo ambiente (indicado por (a) na Figura 4.41), enquanto os recursos para o job oportunista ficaram reduzidos ao mínimo (indicado por (b) na Figura 4.41). Nos momentos em que não existiam processos locais os recursos foram direcionados ao processo remoto, até o limite permitido pelo *OK*. Com a maior disponibilidade de recursos para os processos locais, os tempos de execução dos processos Local-1, Local-2 e Local-3 diminuíram, respectivamente, para *545,12″*, *569,23″* e *583,54″*, o que representou ganhos de 57,52%, 66,88% e 67,62% em relação aos tempos coletados anteriormente. Contribuiu, também, para esse resultado o fato que com mais recursos, cada processo conseguiu terminar sua execução antes que o próximo fosse iniciado, fazendo com que a concorrência entre eles fosse eliminada. Já o tempo do job oportunista foi de *22.620,51″*, representando uma perda de 3,61% em relação ao tempo obtido com o mecanismo inativo.

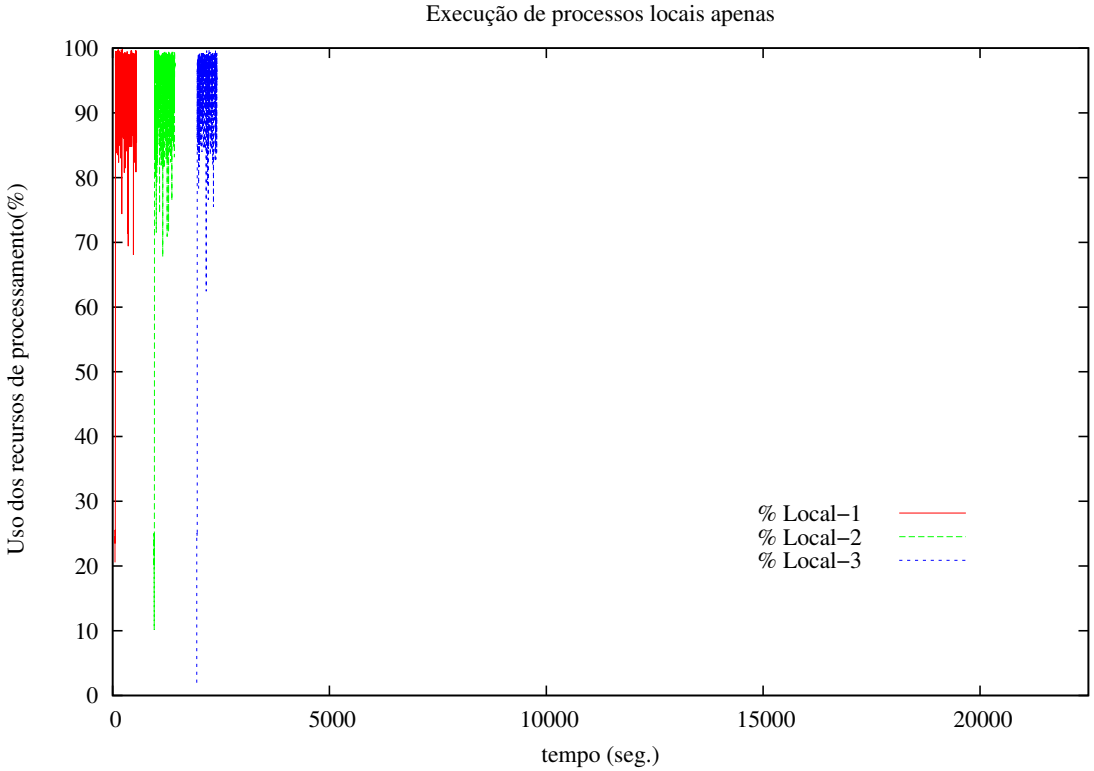

Figura 4.42: Caso 2 - Nó de execução dedicado a processos *locais* e sem mecanismo de controle

Ao retirar da disputa o processo remoto e o mecanismo de controle, a distribuição dos recursos de processamento para os processos locais do *Caso 2* está ilustrado na Figura 4.42. É possível observar que os processos se aproximaram de consumir a capacidade máxima de recursos enquanto estavam executando. Como o intervalo inter-chegada de processos foi maior que o tempo necessário de execução deles, não houve competição por recursos. O tempo de execução de Local-normal_1 foi igual

a *501,45″*, o tempo de Local-2 foi de *486,04″* e o processo Local-3 levou *470,74″* para ser executado. Essa situação representa a situação ideal sob o ponto de vista do usuário, isto é, a estação do usuário não é utilizada para processos que não sejam do próprio dono da máquina, logo esse não é impactado. Por outro lado, os recursos, quando não estão sendo utilizados pelo respectivo usuário, ficam ociosos, mesmo havendo demanda de outros usuários por esses recursos, no caso o processo Remoto-1. Ao se comparar essa situação ideal (Figura 4.42) com aquela em que o mecanismo está ativo (Figura 4.41), verifica-se que a alocação de recursos para os processos locais são similares, diferindo entre si devido aos limites (parâmetros) que são aplicados a utilização do *OK*, o que explica o desempenho inferior desses processos em relação a situação em que não existe processo remoto. Embora os resultados tenham sido inferiores a situação ideal, o mecanismo mitigou o impacto gerado pela utilização da estação do usuário por um processo que não partiu desse nó, que é um dos objetivos de um grid oportunista.

A Figura 4.43 exibe a alocação de recursos para os processos referentes ao *Caso 3* com o mecanismo *inativo*, nessa sequência, o primeiro processo a entrar em execução é um processo Remoto-Pesado e por não concorrer com outros processos locais ou remotos, utiliza o máximo possível de recursos de processamento disponíveis. A medida que novos processos são iniciados (indicados por (a) na Figura 4.43), ocorrem novas disputas e como não há nenhum tipo de restrição, todos os processos poderão concorrer pelos recursos. Verifica-se que o processo oportunista, devido a sua carga, concorre com os demais processos durante todo o período de execução deles, ou seja, impactando-os.

Ao repetir a execução desses processos, mas com o mecanismo *ativo*, chega-se a distribuição de recursos entre processos disponibilizados pela Figura 4.44, nela é possível perceber que, quando há concorrência por recursos, estão alocados aos processos locais, enquanto a distribuição para o processo remoto é mínima. Como consequência, os tempos dos processos locais, sem o mecanismo, que eram de *1.804,56″*, *1.893,76″* e *1830,29″*, foram, respectivamente, para *588,1″*, *499,17″* e *566,55″*.

A distribuição dos recursos para o cenário ideal, em que o mecanismo está desligado e não há processos oportunistas competindo pelos recursos, sob a ótica do usuário desse nó, está presente na Figura 4.45. Observa-se que os processos, além de utilizar praticamente a carga máxima quando estão executando, não competem por recursos entre si, pois acabam antes do próximo iniciar. Sem o mecanismo (Figura 4.43) o job compete inicialmente com o primeiro processo local, fazendo com que esse tenha seu término postergado e supere o tempo de chegada do segundo processo. Nesse ponto, há três processos competindo e não dois, conforme a situação ideal, o que irá impactar, também o tempo de execução do processo local-2. Ao introduzir o mecanismo foi possível, como nos casos anteriores, aproximar a distribuição de

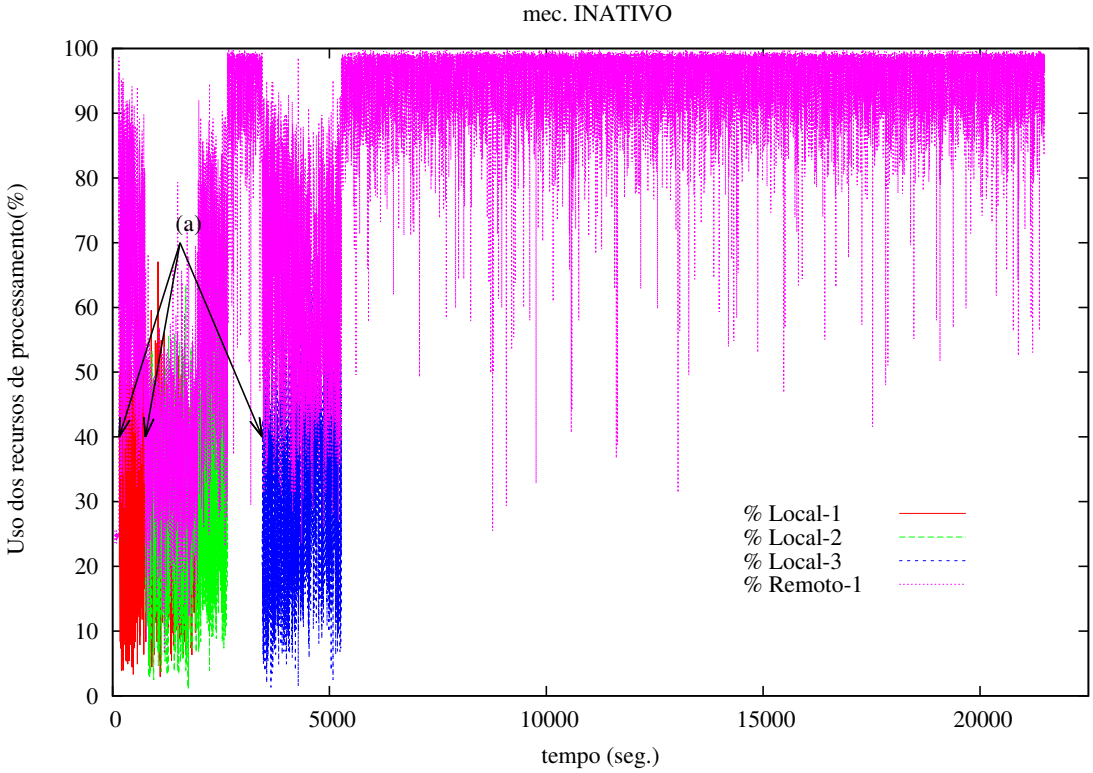

Figura 4.43: Caso 3 - Competição por recursos entre 1 processo local pesado e 3 processos remotos normais com o mecanismo INATIVO

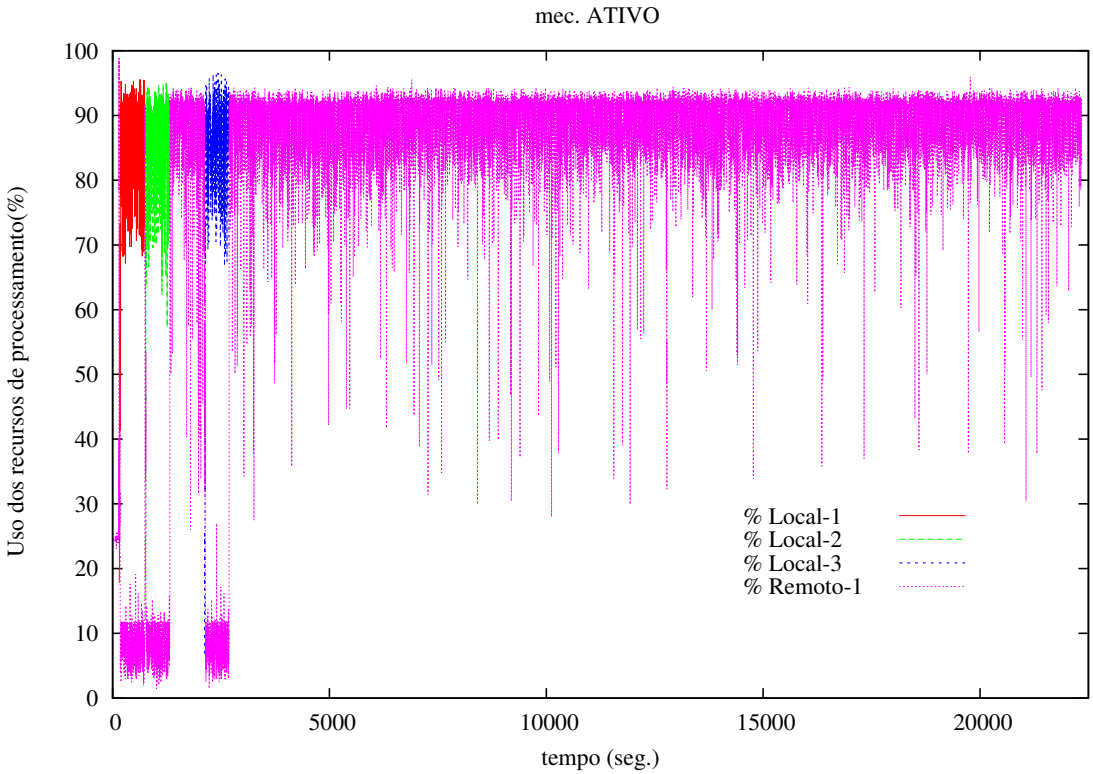

Figura 4.44: Caso 3 - Competição por recursos entre 1 processo local pesado e 3 processos remotos normais com o mecanismo ATIVO

recursos da situação em que há processo oportunista com a situação considerada ideal.

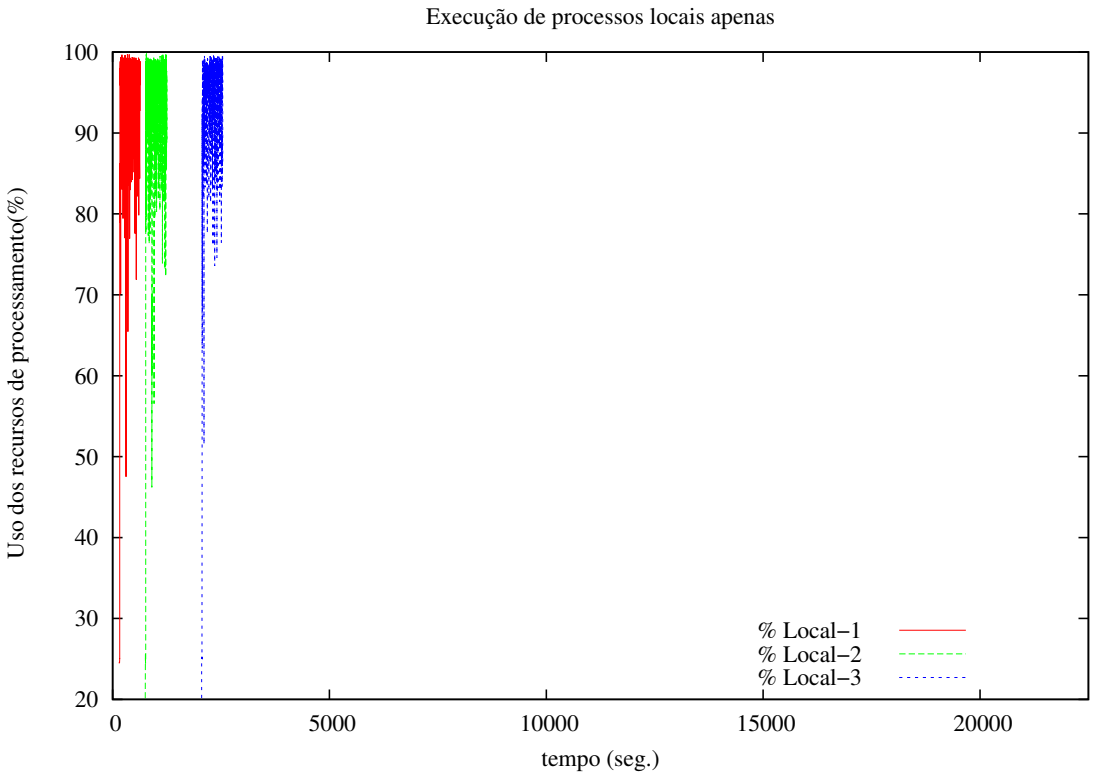

Figura 4.45: Caso 3 - Nó de execução dedicado a processos *locais* e sem mecanismo de controle

Nos casos anteriores, os processos locais só disputaram recursos entre si quando concorreram com os processos remotos nas situações em que o mecanismo estava inativo. A próxima configuração de processos (*Caso 4*) faz a execução de parte dos processos locais serem concorrentes, averiguando os impactos que a demanda de processos oportunistas causa naqueles processos e como o mecanismo atuou sobre essa situação. A Figura 4.46, que mostra a situação ideal, confirma a disputa por recursos entre os processos locais, que iniciaram no momento em que o processo Local-2 entrou em execução (indicado por (a)), e se agravaram com a chegada de Local-3 (indicado por (b)). Quando o primeiro processo terminou sua execução (indicado por (c)), os demais processos tiveram acesso a mais recursos, embora a condição de corrida tenha permanecido entre eles. Os tempos apurados foram: Local-1 = *807,31″*, Local-2 = *1.252,46″* e Local-3 = *1.177,69″*.

Ao se introduzir os processos oportunistas, os tempos dos três processos locais foram, respectivamente, *1.721,40″*, *1.875,18″* e *1.888,28″*. Enquanto os tempos obtidos para os processos remotos foram de *1.310,17″* para o primeiro e *1.252,70″* para o segundo. A Figura 4.47, que apresenta a distribuição dos recursos para esses processos, mostra que o primeiro processo remoto executou sozinho até que o

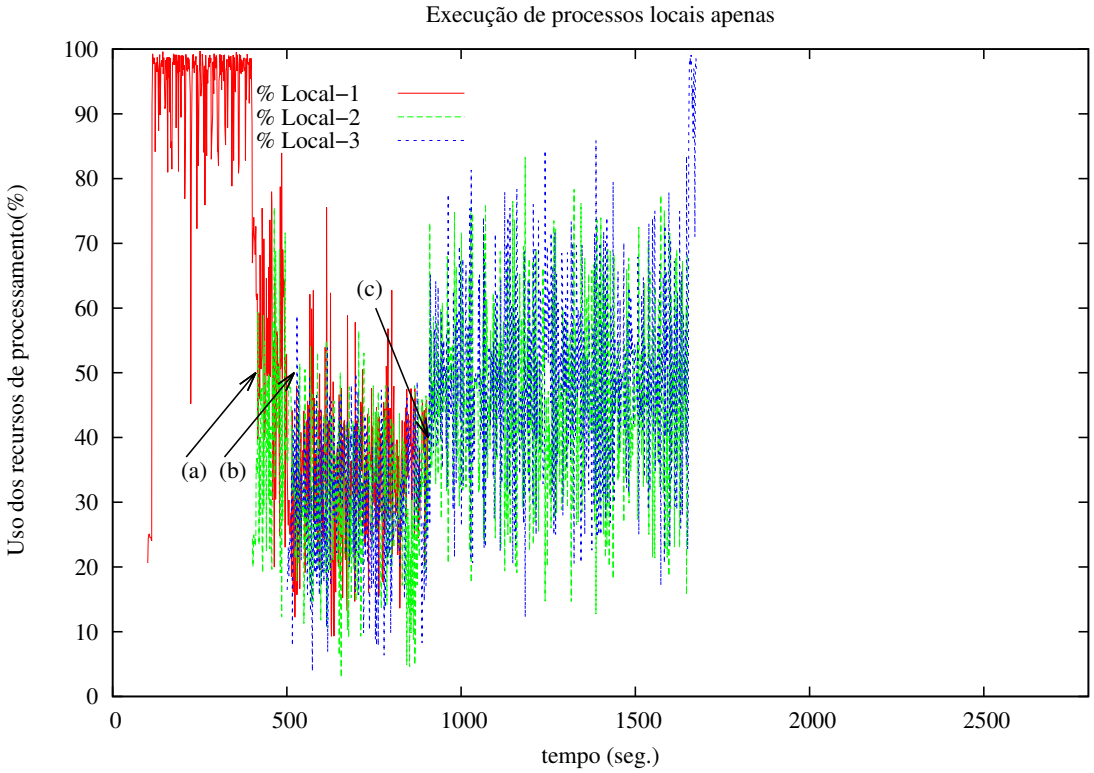

Figura 4.46: Caso 4 - Nó de execução dedicado a processos *locais* e sem mecanismo de controle

primeiro processo local foi inicializado (indicado por (a)), contribuindo para que o total de recursos para cada processo fosse a metade da capacidade do nó, ou seja, o processo remoto impactou o processo local. Nos instantes em que mais processos entraram em execução (indicado por (b) na Figura 4.47), novamente os recursos foram divididos de forma igual entre os processos, tornando os recursos ainda mais escassos. Essa situação só foi amenizada a partir dos momentos em que processos foram terminando (indicado por (c-remoto) e (d-locais)), o que permitiu que mais recursos fossem liberados, e por consequência, disponibilizados para os processos, mas a disputa se manteve e os processos oportunistas continuaram a impactar os processos dos prioritários. Após o término do último processo local, o processo oportunista remanescente recebeu todos os recursos disponíveis do nó.

Repetindo o caso anterior com o mecanismo *ativo*, os seguintes tempos foram obtidos para os processos locais: Local-1 *1.015,37″* (melhora de 41,01%), Local-2 *1.399,14″* (melhora de 25,38%), Local-3 *1.342,28″* (melhora de 28,72%). Avaliando a Figura 4.48, que exibe a distribuição dos recursos para os processos, constata-se que após o início do processo Local-1, o mecanismo verifica que processo(s) local(is) e remoto(s) está(ão) concorrendo por recursos (indicado por (a) na Figura 4.48), e este deve ser restringido até que nenhum processo local esteja presente no nó (intervalo entre (b) e (c)). Dessa forma, com a chegada dos outros processos locais

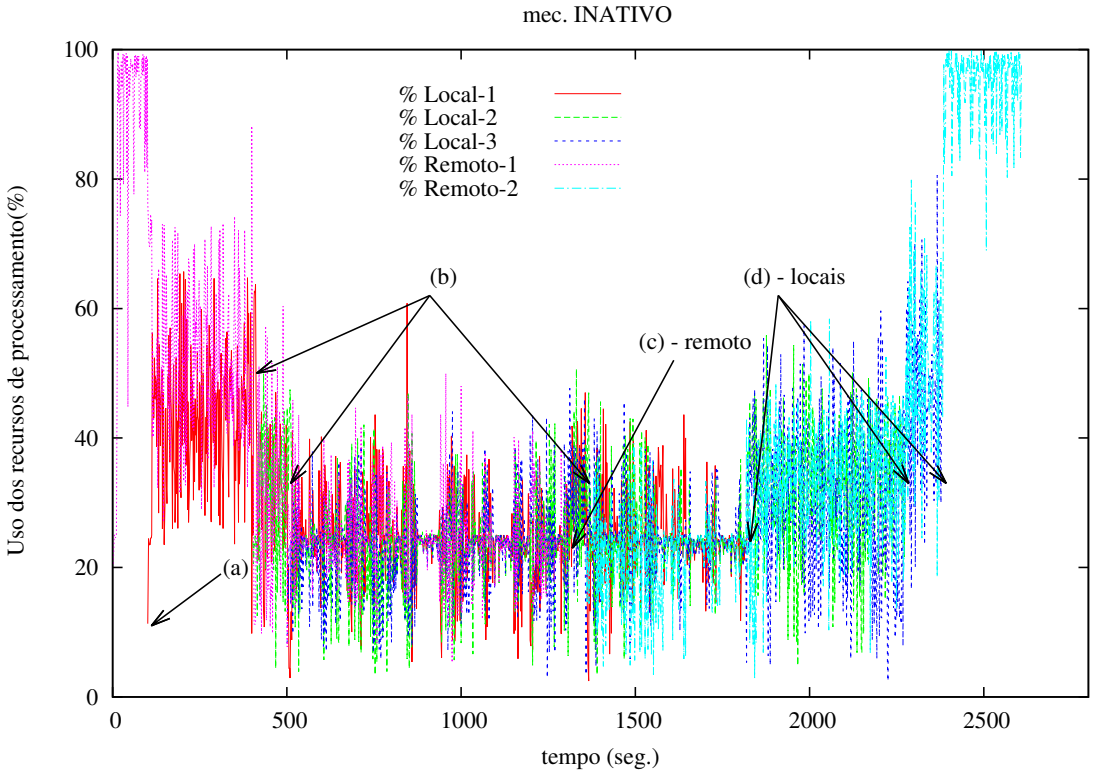

Figura 4.47: Caso 4 - Competição por recursos entre 3 processos locais e 2 processos remotos normais com o mecanismo INATIVO

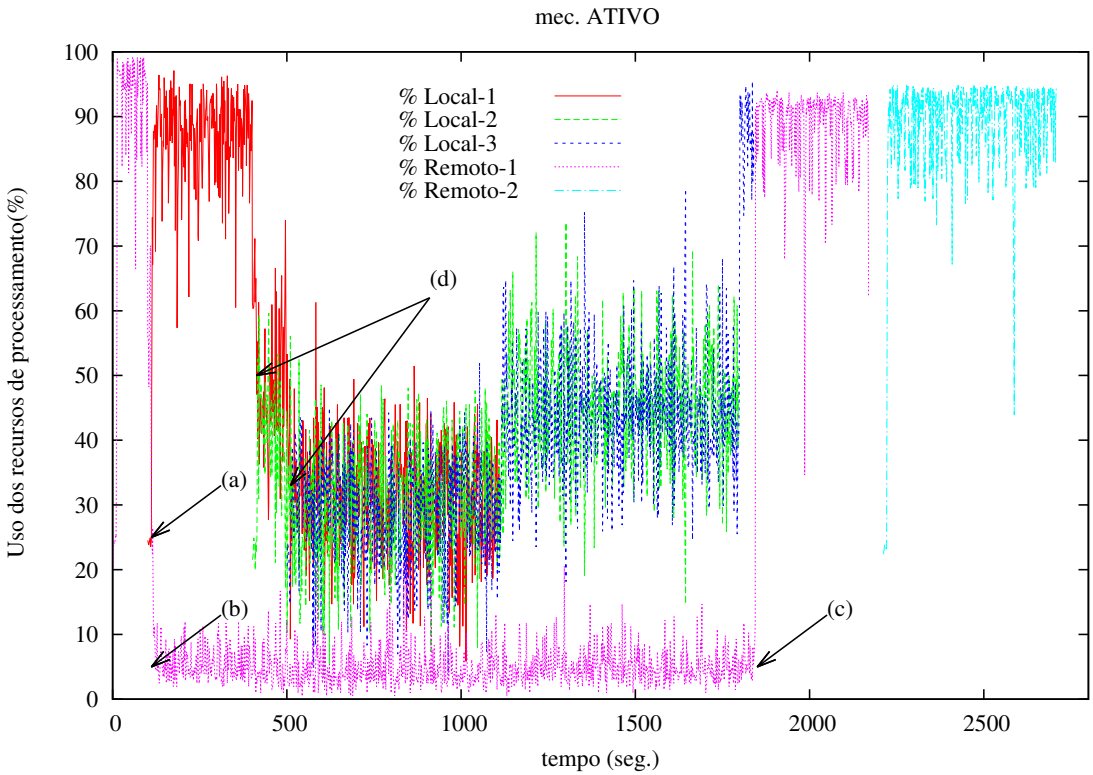

Figura 4.48: Caso 4 - Competição por recursos entre 3 processos locais e 2 processos remotos normais com o mecanismo ATIVO

(indicado por (d)), a disputa ocorreu sem a interferência do job, pois os recursos destinados a ele já estava delimitado, e a maior parte dos recursos foi distribuída apenas aos processos locais. Com isso, o tempo de execução destes processos, se comparado com a situação em que o mecanismo estava inativo, diminuiu.

Em relação ao desempenho dos processos oportunistas, o tempo de execução de Remoto-1 foi de *2.171.35″*, o que representou uma perda de 65,73%, o que é explicado pela restrição imposta pelo mecanismo, enquanto o tempo do segundo processo remoto foi de *498,33″*, o que representou um melhora de 60,21%. Essa melhora é devida pelo fato que o *OK*, antes de colocar em execução um processo remoto, verifica se já não há um outro job em execução, se existir, o processo oportunista requisitante é colocado em espera até que o nó não tenho nenhum processo desse tipo. Essa foi a situação que ocorreu, pois como já existia um job em execução, o processo Remoto-2 foi colocado em fila de espera até que o outro processo remoto terminasse sua execução. Quando Remoto-2 iniciou sua execução, não haviam outros processos para competir, portanto foi possível utilizar a quantidade de recursos máxima permitida pelo *OK*, diminuindo o tempo de execução. Embora tenha existido ganho no que se refere ao tempo de execução desse job, o tempo de retorno que era de *2.608,45″* na situação em que o mecanismo estava *inativo* foi para *2.708,29″* na situação com o mecanismo *ativo*, isto é, houve queda de performance.

4.1.4 Avaliação dos experimentos

Com o término desses experimentos é possível avaliar as situações em que a utilização do *OK* pode trazer ganhos para a adoção de um solução com *grid oportunista* e avaliar se o mesmo pode atenuar algumas limitações (conforme mostrado no capítulo 2) inerentes a esse tipo de solução.

Para os casos em que as aplicações são *IO-bound* (seção 4.1.2) o mecanismo se mostrou pouco eficiente. Esse comportamento já era esperado e foi confirmado pelos experimentos, haja vista que o mecanismo está focado no(s) processador(es), não tendo atuação direta sobre as primitivas de leitura e escritas de dados do nó de execução. Os resultados mostraram que o nó executor do *OK* ao receber processo(s) *IO-bound* tem praticamente o mesmo desempenho que um nó de um grid oportunista sem o mecanismo de limitação, portanto o overhead ocasionado pela adoção do mecanismo torna viável a utilização da solução proposta nesse trabalho para aplicações desse tipo. O fato de não impactar negativamente os processos *IO-bound* locais, permite que o mecanismo seja adotado em todos os nós de um grid, e quando surgir uma situação de concorrência por recursos de processamento, os impactos gerados pelos processos remotos serão mitigados, ou seja, os processos dos usuários terão um desempenho igual ou melhor com o mecanismo ativo do que se não houvesse tal

mecanismo.

Quando as aplicações eram *CPU-bound* (seção 4.1.1), os resultados obtidos mostram que a adoção do *OK* trará ganhos nas condições em que há intensa disputa por recursos (cenários com o valor de ρ igual a 50% e 90%), que são as situações em que os jobs impactam os processos dos usuários. Nos outros cenários (ρ igual a 10%), o desempenho do mecanismo diminuiu, aproximando-se de zero, devido a baixa concorrência entre os processos. Sendo assim, o nó do *OK*, no pior dos casos, terá desempenho similar a um grid sem o mecanismo, para os casos em que não há disputa, mas a partir que essa situação se altera, o *OK* se mostra mais eficiente.

Ao utilizar o *OK* com aplicações reais (4.1.3), foi possível observar, além dos ganhos, a mitigação dos impactos causados aos processos locais através da restrição/liberação de recursos imposta aos processos oportunistas. A capacidade de determinar o momento e a quantidade de recursos que serão alocados aos processos oportunistas são grandes vantagens em relação a nós sem esse mecanismo, pois aqueles podem ser utilizados em tempo integral, enquanto a utilização destes para processar jobs, devido ao impacto gerado aos usuários, fica limitada a aceitação desses impactos, ou sua utilização fica limitada aos momentos em que não há utilização dos nós.

Os experimentos mostraram que o *OK* atua de maneira transparente para o usuários e sem a necessidade de um controlador externo para o nó executor. Além disso, a alocação de recursos, dependendo da demanda, pode ser alterado em qualquer momento, garantindo que nos momentos em que processos locais estão requisitando recursos de processamento, eles serão priorizados, conforme esses processos diminuem suas demandas, os recursos são disponibilidados para os processos oportunistas.

Capítulo 5

Conclusões e Trabalhos Futuros

Nesse trabalho foi a apresentado um *grid* oportunista de *workstations*, tendo como principal característica a capacidade de limitar os recursos para os processos provenientes desse *grid*, através de mecanismos de restrições que executam no próprio nó de execução, portanto de forma descentralizada. Esse comportamento foi obtido através do desenvolvimento de *daemons* específicas para esse fim, sendo que a principal características destas é avaliar a utilização da capacidade de processamento daquele nó e dependendo do resultado dessa análise executar a ação de liberar ou restringir recursos para os *jobs*. A aplicabilidade dessas ações foi possível devido ao desenvolvimento de rotinas capazes de manipular a estrutura de gerenciamento de processos de cada nó e, com isso, podendo modificar o *time-slice* que fora definido pelo escalonador do sistema operacional. Dessa forma, o controle da quantidade de recursos de processamento alocados para os *jobs* naquele nó é realizado dinamicamente pelo *kernel*.

Essa solução, denominada *OK*, foi planejada em um modelo de três camadas: *(i)* submissão: responsável pela submissão do *job*; *(ii)* alocação: faz a alocação do nó executor para o *job*, e; *(iii)* execução: realiza o processamento e aplica as restrições. A implementação consistiu na utilização da ferramenta de submissão de *jobs* *Torque*, com algumas alterações, para desempenhar as funções comuns de todo *grid* (gerenciar a fila de *jobs*, coletar dados e encaminhar os *jobs* para execução), e para as funções específicas (controle de recursos) foram desenvolvidos rotinas próprias.

Foram realizados alguns experimentos para verificar o comportamento dos procedimentos implementados e avaliar o desempenho do mecanismo de restrição quando aplicado a um ambiente em que processos locais e *jobs* compartilham os mesmos recursos, devendo garantir a *non-intrusiveness* dos processos oportunistas. As aplicações utilizadas nos ensaios tiveram seus códigos-fontes e configurações inalteradas, atendendo a um dos requisitos do *OK*.

Os resultados demonstraram que, embora os melhores resultados tenham sido alcançados nas aplicações cpu-bound (principalmente nas ocasiões com grande con-

corrência entre processos locais e remotos), a utilização do mecanismo em processos io-bound não degradou o desempenho desses processos. Dessa forma, o mecanismo pode ser ativado em todos todos os nós, independentemente do tipo do job executado, nos momentos em que os processos oportunistas demandarem recursos de processamento e estejam concorrendo com processos locais, os recursos para aqueles serão restringidos, mitigando os possíveis impactos causados; caso os processos remotos não exijam esse tipo de recurso o resultado da ação do mecanismo sobre esses processos será inócuo.

A atuação do mecanismo ocorreu de maneira transparente para os usuários, de forma descentralizada e principalmente em período integral, pois a performance interativa da estação com o usuário fica próxima de uma workstation dedicada, a medida que os recursos alocados aos processos remotos podem ser rapidamente diminuídos e destinados aos processos dos usuários. Além disso, é possível, através de parâmetros, definir os limites mínimos e máximos de recursos que serão alocados aos jobs, maximizando a utilização desses recursos, mesmo nas situações que o nó está sendo utilizado por um usuário.

Embora o *OK* tenha se mostrado eficiente na mitigação de impactos gerados pelos processos remotos sobre os processos locais, algumas aplicações com restrições de tempo podem ser consideradas inviáveis (inelegíveis) de serem processadas no grid. Pois a solução, conforme está apresentada, não define porções de recursos alocadas *exclusivamente* para os processos oportunistas. É com esse objetivo que se propõe, como trabalho futuro, a modificação do *OK* para que requisitos de *QoS* (*Quality of Service*) sejam aceitos, possibilitando, dessa forma, aumentar o número de aplicações que podem utilizar o grid e contornar um dos grandes gargalos de um grid oportunista, que é falta de garantia sobre o tempo de retorno para as aplicações que executam nesse tipo de *grid*.

Para atender esse objetivo, será necessário aplicar o mesmo mecanismo de restrição descrito no presente trabalho, com a diferença de que os processos que poderão sofrer restrições, a nível de *kernel*, são os locais. De uma maneira geral, todos os pontos levantados (*configuração nas máquinas locais (nós), mecanismos de coleta de informações e monitoramento das workstations, Middleware, etc*) anteriormente serão mantidos, as alterações necessárias são, basicamente:

- Submissão do(s) *job(s)*: ao submeter um *job* a aplicação deve informar qual será o requisito mínimo (expresso em valor percentual) de recursos de processamento necessário para a execução desse processo;

- Auto-ajuste dos processos do *Grid*: o auto-ajuste será feito em virtude do requisito de processamento informado no momento da submissão, logo, os processos que serão controlados e limitados, em tempo de execução, localmente,

sem a necessidade de controlador centralizado e no nível do kernel, serão os processos *locais*.

Ao término do desenvolvimento desse mecanismo, surgem novas situações ou mesmo problemas que antes não existiam. Entre estes, destacam-se:

- Definição de qual é o percentual máximo de recursos que uma aplicação pode alocar, sem causar grandes impactos aos usuários;
- Escolha de quais processos locais são *suscetíveis* ao mecanismo;
- Identificação do impacto na usabilidade do nó que está sendo aplicado os requisitos de *QoS*.

Outro trabalho que pode aprimorar o desempenho do *OK* é a implementação de um método que seja sensível ao contexto, ou seja, a política de restrição (valor dos parâmetros) irá variar dependendo do tipo dos processos presentes no ambiente em cada momento. Para isso será necessário definir tais políticas, identificar as situações que melhor se adaptam a determinada política e inseri-las no *OK*.

A análise do impacto que a redefinição dos valores dos parâmetros (L+,L-,R_-max , R_mim, etc) e a melhoria/substituição da rotina *Ajustar()* pode trazer para a presente solução é um importante trabalho que pode ser realizado, já que essas configurações impactam diretamente os resultados obtidos com a utilização do mecanismo de restrição.

A migração de versão sistema operacional está sendo uma realidade, e por consequência a versão do *kernel* também esta sendo atualizada, a qual é diferente da versão de *kernel* utilizada no mecanismo. Sendo assim, é necessário aplicar/portar as modificações realizadas no *kernel* anterior para o atual e/ou encapsular o mecanismo em um módulo [62] que poderá ser carregado, instanciado e atualizado independentemente da versão do kernel e sistema operacional.

Referências Bibliográficas

[1] MARQUES, O., DRUMMOND, T. "Building a software infrastructure for computational science applications: lessons and solutions". In: *Second international workshop on Software engineering for high performance computing system applications (SE-HPCS '05)*, pp. 40–44, New York, NY, USA, 2005. ACM. ISBN: 1-59593-117-1. doi: http://doi.acm.org/10.1145/1145319.1145332.

[2] SIERRA, G., MCNAUGHT, J. "Extracting semantic clusters from the alignment of definitions". In: *18th conference on Computational linguistics*, pp. 795–799, Morristown, NJ, USA, 2000. Association for Computational Linguistics. doi: http://dx.doi.org/10.3115/992730.992761.

[3] FOSTER, I. "What is the Grid? A Three Point Checklist". . Disponível em http://dlib.cs.odu.edu/WhatIsTheGrid.pdf. Acessado em Julho de 2012.

[4] BRASILEIRO, F., MIRANDA, R. "The OurGrid Approach for Opportunistic Grid Computing", *First EELA-2 Conference*, pp. pp. 11–19., 2009.

[5] ANDERSON, T., CULLER, D., PATTERSON, D. "A case for NOW (Networks of Workstations)", *Micro, IEEE*, v. 15, n. 1, pp. 54–64, 1995. ISSN: 0272-1732. doi: 10.1109/40.342018.

[6] ZHOU, D., LO, V. "WaveGrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system". In: *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, p. 10 pp., April 2006. doi: 10.1109/IPDPS.2006.1639267.

[7] SINGH, A., AWASTHI, L. "Performance comparisons and scheduling of load balancing strategy in Grid Computing". In: *International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, pp. 438 –443, April 2011. doi: 10.1109/ETNCC.2011.6255939.

[8] BOUGUERRA, M., KONDO, D., TRYSTRAM, D. "On the Scheduling of Checkpoints in Desktop Grids". In: *11th IEEE/ACM International Sym-*

posium on Cluster, Cloud and Grid Computing (CCGrid '11), pp. 305 –313, May 2011. doi: 10.1109/CCGrid.2011.63.

[9] BATISTA, D. M., DA FONSECA, N. L. S., MIYAZAWA, F. K., et al. "Self-adjustment of resource allocation for grid applications", *The International Journal of Computer and Telecommunications Networking*, v. 52, n. 9, pp. 1762–1781, 2008. ISSN: 1389-1286.

[10] RAMACHANDRAN, K., LUTFIYYA, H., PERRY, M. "Decentralized Resource Availability Prediction for a Desktop Grid". In: *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid '10)*, pp. 643 –648, May. 2010. doi: 10.1109/CCGRID.2010.54.

[11] HE, T., CHEN, S., KIM, H., et al. "Scheduling Parallel Tasks onto Opportunistically Available Cloud Resources". In: *5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 180 –187, June 2012. doi: 10.1109/CLOUD.2012.15.

[12] CHIEN, A., CALDER, B., ELBERT, S., et al. "Entropia: Architecture and performance of an enterprise desktop grid system", *Journal of Parallel and Distributed Computing*, v. 63, n. 5, pp. 597–610, 2003.

[13] SMITH, J., NAIR., R. "The Architecture of Virtual Machines", *COMPUTER,*, p. 32–38, 2005.

[14] CASTRO, H., ROSALES, E., VILLAMIZAR, M., et al. "UnaGrid: On Demand Opportunistic Desktop Grid". In: *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID '10)*, pp. 661–666, Washington, DC, USA, 2010. IEEE Computer Society. ISBN: 978-0-7695-4039-9. doi: http://dx.doi.org/10.1109/CCGRID.2010.79.

[15] DORNEMANN, K., BOSCHANSKI, U., ZEISS, A., et al. "Integrating Virtual Execution Environments into Peer-to-Peer Desktop Grids". In: *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 333 –340, Feb. 2012. doi: 10.1109/PDP.2012.39.

[16] SAAD, W., ABBES, H., CERIN, C., et al. "A Self-Configurable Desktop Grid System On-Demand". In: *Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pp. 196 –203, Nov. 2012. doi: 10.1109/3PGCIC.2012.6.

[17] FERREIRA, D., ARAUJO, F., DOMINGUES, P. "libboincexec: A Generic Virtualization Approach for the BOINC Middleware". In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pp. 1903 – 1908, May 2011. doi: 10.1109\IPDPS.2011.349.

[18] WANG, D., GONG, B. "SUCSI: A Light-Weight Desktop Grid System Using Virtualization for Application Sandboxing". In: *International Conference on Network Computing and Information Security (NCIS)*, v. 1, pp. 352 –356, May 2011. doi: 10.1109/NCIS.2011.78.

[19] ZHANG, Y., LI, Y., ZHENG, W. "Using User-Level Virtualization in Desktop Grid Clients for Application Delivery and Sandboxing". In: *Fourth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pp. 289 – 293, Dec. 2011. doi: 10.1109/PAAP.2011.44.

[20] HE, S., GUO, L., GHANEM, M., et al. "Improving Resource Utilisation in the Cloud Environment Using Multivariate Probabilistic Models". In: *5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 574 –581, June 2012. doi: 10.1109/CLOUD.2012.66.

[21] MARSHALL, P., KEAHEY, K., FREEMAN, T. "Improving Utilization of Infrastructure Clouds". In: *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '11)*, pp. 205 –214, May 2011. doi: 10.1109/CCGrid.2011.56.

[22] HAN, R., GUO, L., GHANEM, M., et al. "Lightweight Resource Scaling for Cloud Applications". In: *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12)*, pp. 644 – 651, May 2012. doi: 10.1109\CCGrid.2012.52.

[23] MENG, X., ISCI, C., KEPHART, J., et al. "Efficient resource provisioning in compute clouds via VM multiplexing". In: *7th international conference on Autonomic computing*, ICAC '10, pp. 11–20, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0074-2. doi: 10.1145/1809049.1809052.

[24] HO, Y., LIU, P., WU, J.-J. "Server Consolidation Algorithms with Bounded Migration Cost and Performance Guarantees in Cloud Computing". In: *Fourth IEEE International Conference on Utility and Cloud Computing (UCC)*, pp. 154 –161, Dec. 2011. doi: 10.1109/UCC.2011.30.

[25] CHAKRAVARTI, A., BAUMGARTNER, G., LAURIA, M. "The organic grid: self-organizing computation on a peer-to-peer network", *IEEE Tran-*

sactions on Systems, Man and Cybernetics, Part A: Systems and Humans, v. 35, n. 3, pp. 373 – 384, May 2005. ISSN: 1083-4427. doi: 10.1109/TSMCA.2005.846396.

[26] TABATA, T., HAKOMORI, S., YOKOYAMA, K., et al. "Controlling CPU Usage for Processes with Execution Resource for Mitigating CPU DoS Attack". In: *International Conference on Multimedia and Ubiquitous Engineering (MUE '07)*, pp. 141 –152, April 2007. doi: 10.1109/MUE.2007.111.

[27] YIGITBASI, N., SONMEZ, O., IOSUP, A., et al. "Performance Evaluation of Overload Control in Multi-cluster Grids". In: *12th IEEE/ACM International Conference on Grid Computing (GRID)*, pp. 173 –180, Sept. 2011. doi: 10.1109/Grid.2011.30.

[28] SHARMA, B., PRABHAKAR, R., LIM, S., et al. "MROrchestrator: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters". In: *5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 1 –8, June 2012. doi: 10.1109/CLOUD.2012.37.

[29] ZHANG, Y., KOELBEL, C., COOPER, K. "Hybrid Re-scheduling Mechanisms for Workflow Applications on Multi-cluster Grid". In: *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, pp. 116 –123, May. 2009. doi: 10.1109/CCGRID.2009.60.

[30] CELAYA, J., MARCHAL, L. "A Fair Decentralized Scheduler for Bag-of-tasks Applications on Desktop Grids", *LIP Research Report RR-LIP 2010-07*, pp. 1–23, 2010.

[31] ANGLANO, C. "Fair Scheduling of General-Purpose Workloads on Workstation Clusters", *Cluster Computing*, v. 5, n. 1, pp. 87–96, 2002. ISSN: 1386-7857. doi: http://dx.doi.org/10.1023/A:1012752923793.

[32] MASNIDA HUSSIN, YOUNG CHOON LEE, A. Y. Z. "Dynamic Job-Clustering with Different Computing Priorities for Computational Resource Allocation", *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid '10)*, pp. 589–590, 2010.

[33] TANENBAUM, A. S. *Computer Networks - 4th Edition, ISBN 0-13-066102-3*. Pearson Education, 2003.

[34] HALIM, R., ABIDIN, S., LATIP, R. "Grid task scheduling in P2P Desktop Grids". In: *IEEE Business Engineering and Industrial Applications Colloquium (BEIAC)*, pp. 150 –155, April 2012. doi: 10.1109/BEIAC.2012.6226041.

[35] CHOI, S., KIM, H., BYUN, E., et al. "Characterizing and Classifying Desktop Grid". In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGRID '07)*, pp. 743 –748, May 2007. doi: 10.1109/CCGRID.2007.31.

[36] HAN ZHAO, X. L., LI, X. "A taxonomy of peer-to-peer desktop grid paradigms", *Cluster Computing*, v. 14, Issue 2, pp. 129–144, 2011.

[37] BARHAM, P. E. A. "Xen and the art of virtualization", *Proceedings of SOSP '03. New York, USA: ACM*, p. 164–177, 2003.

[38] CITRIX. "Home of the Xen hypervisor. 2009." . http://www.xenl.org˙Acessado em Setembro de 2012.

[39] VMWARE. "Virtualization Software for Desktops, Servers & Virtual Machines for a Private Cloud." . http://www.vmware.org˙ Acessado em Setembro de 2012.

[40] ORACLE. "Virtual Box". . https://www.virtualbox.org. Acessado em Outubro de 2013.

[41] CONTEINERS, O. L. "OpenVZ". . https://www.openvz.org. Acessado em Outubro de 2013.

[42] DOS REIS, V. Q. *Um estudo sobre reserva de recursos computacionais no nível do usuário.* Tese de Doutorado, Pontifícia Universidade Católica do Rio de Janeiro - PUC-RIO, 2010.

[43] M. LITZKOW, M. L., MUTKA, M. "Condor — A Hunter of Idle Workstations". In: *8th International Conference of Distributed Computing Systems*, 1988.

[44] HUEDO, E., MONTERO, R., LLORENTE, I. "Experiences on adaptive grid scheduling of parameter sweep applications", *12th Euromicro Conference on Parallel, Distributed and Network - Based Processing*, p. 28–33, 2004.

[45] ALLEN, G., ANGULO, D., FOSTER, I., et al. "The cactus worm: experiments with dynamic resource discovery and allocation in a grid environment", *Int. J. High Performance Comput. Appl.*, v. 15, pp. 345–358, 2001.

[46] BERMAN, F., WOLSKI, R., CASANOVA, H., et al. "Adaptive computing on the grid using AppLeS", *IEEE Trans. Parallel Distribut*, v. 14, pp. 369–382., 2003.

[47] VADHIYAR, S., DONGARRA, J. "A performance oriented migration framework for the grid", *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '03)*, p. 130–137, 2003.

[48] MARLETTA, A. "CPU Usage Limiter for Linux". . Disponível em http://cpulimit.sourceforge.net, 2013. Acessado em Fevereiro de 2013.

[49] GOYAL, P.; GUO, X. V. H. M. "A hierarchical cpu scheduler for multimedia operating systems. Readings in multimedia computing and networking,", *Morgan Kaufmann Publishers Inc., San Francisco, CA, USA,*, p. 491–505, 2001.

[50] SCHULZ, S., BLOCHINGER, W., HELD, M., et al. "COHESION - A microkernel based Desktop Grid platform for irregular task-parallel applications", *Future Gener. Comput. Syst.*, v. 24, pp. 354–370, 2008. ISSN: 0167-739X. doi: 10.1016/j.future.2007.06.005.

[51] WANG, Y.-C.; LIN, K.-J. "Implementing a general real-time scheduling framework in the red-linux real-time kernel." *Proceedings of RTSS '99.*, p. 246–255, 1999.

[52] CHU, H.-H.; NAHRSTEDT, K. *Proceedings of IDMS '97. London, UK: Springer- Verlag, 1997*, p. 153–162., 1997.

[53] CHANG, F.; ITZKOVITZ, A. K.-V. "User-level resource- constrained sandboxing." *Proceedings of WSS '00. Berkeley, USA: USENIX Association*, p. 3–3, 2000.

[54] REIS, V. Q. DOS; CERQUEIRA, R. "A tool for isolating performance in general-purpose operating systems." *Proceedings of MGC '08. New York, USA ACM Press*, p. p. 1–6., 2008.

[55] SCHOPF, J. *Ten actions when grid scheduling, in: Grid Resource Management: State of the Art and Future Trends.* J.M. Schopf, 2003.

[56] CLUSTER RESOURCES INC., PROVO, U. "TORQUE Resource Manager". . Disponível em at http://www.adaptivecomputing.com/products/open-source/torque/, 2013. Acessado em Janeiro, 2013.

[57] VLADISHEV, A. "Zabbix". . disponivel em http://www.zabbix.com/, 2010. Acessado em Setembro de 2010.

[58] COMPUTING, P. "Load Sharing Facility (LSF)". . http://www.platform.com/-workload-management/high-performance-computing/lp, set 2010. Acessado em setembro de 2010.

[59] ORACLE. “Oracle Grid Engine”. . Disponível em http://www.oracle.com/-us/products/tools/oracle-grid-engine-075549.html, set 2010. Acessado em setembro de 2010.

[60] BOB SMITH, JOHN HARDIN, G. P. B. P. *Linux Appliance Design A Hands-On Guide to Building Linux Applications*. No Starch Press, 2007.

[61] GIACOBBI, G. “The GNU Netcat”. . Disponível em http://netcat.sourceforge.net/, Sept 20102. Acessado em Setembro de 2012.

[62] DANIEL P. BOVET, M. C. P. *Understanding the Linux Kernel*. O’Reilly, 2006. ISBN-13: 978-0-596-00565-8.

[63] LOVE, R. *Linux Kernel Development*. Addison-Wesley, 2010. ISBN-13: 978-0-672-32946-3.

[64] REDHAT. “Red Hat Documentation”. . Disponpivel em http://docs.redhat-.com/, Sept 2012. Acessado em Setembro de 2012.

[65] CENTOS. “The Community ENTerprise Operating System”. . http://www.-centos.org/, 2012. Acessado em Setembro de 2012.

[66] HAT, R. “Red Hat Enterprise Linux”. . http://www.redhat.com/-products/enterprise-linux/, 2012. Acessado em Setembro de 2012.

[67] LINUX, S. “Scientific Linux”. . http://www..scientificlinux.org/, 2012. Acessado em Setembro de 2012.

[68] WATERLAND, A. “Stress”. . Disponível em http://weather.ou.edu/ apw/-projects/stress/, 2013. Acessado em Janeiro de 2013.

[69] BRITO, N. L. C. *Otimização Multiobjetivo em Redes de Filas*. Tese de Doutorado, Universidade Federal de Minas Gerais - UFMG, 03 2013. Disponível em: <http://hdl.handle.net/1843/BUOS-974FY2>.

[70] JAIN, R. *The art of computer systems performance analysis*. John Wiley & Sons, Inc., 1991.

[71] BARRERA, D. “Linux Test Project”. . http://ltp.sourceforge.net/, 2012. Acessado em Setembro de 2012.

[72] LTD., C. M. G. “IMEX - Three-Phase, Black-oil Reservoir Simulator”. . http-://www.cmgl.ca/soft-imex. Acessado em Setembro de 2012.