UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

O Impacto da Métrica e do Escalonador sobre a Performance dos Supercomputadores Paralelos

Josilene Aires Moreira

Campina Grande - PB
Abril - 2003

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

O Impacto da Métrica e do Escalonador sobre a Performance dos Supercomputadores Paralelos

Josilene Aires Moreira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal de Campina Grande, como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Redes de Computadores e Sistemas Distribuídos

Walfredo da Costa Cirne Filho

(Orientador)

Campina Grande - PB
Abril - 2003

MOREIRA, Josilene Aires

M835I

O Impacto da Métrica e do Escalonador sobre a Performance dos Supercomputadores Paralelos

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Maio de 2003.

88p. Il.

Orientador: Walfredo da Costa Cirne Filho

Palavras Chave:

1. **Arquitetura de Computadores**
2. **Processamento Paralelo**
3. **Avaliação de Desempenho de Supercomputadores**
4. **Escalonamento de Supercomputadores**

CDU – 681.322

"O IMPACTO DA MÉTRICA E DO ESCALONADOR SOBRE A PERFORMANCE DOS SUPERCOMPUTADORES PARALELOS"

JOSILENE AIRES MOREIRA

DISSERTAÇÃO APROVADA EM 16.05.2003

Walfredo da Costa Cirne Filho

PROF. WALFREDO DA COSTA CIRNE FILHO, Ph.D
Orientador

PROF. ELMAR UWE KURT MELCHER, Dr.
Examinador

PROF. CESAR AUGUSTO F. DE ROSE, Dr.
Examinador

CAMPINA GRANDE – PB

"Tudo tem o seu tempo determinado, e há tempo para todo propósito debaixo do céu:
Há tempo de nascer e tempo de morrer;
Tempo de plantar e tempo de arrancar o que se plantou;
Tempo de matar e tempo de curar; tempo de derribar e tempo de edificar;
Tempo de chorar e tempo de rir; tempo de prantear e tempo de saltar;
Tempo de espalhar pedras e tempo de a-juntar pedras; tempo de abraçar e tempo de afastar-se de abraçar;
Tempo de buscar e tempo de perder;
Tempo de guardar e tempo de deitar fora;
Tempo de rasgar e tempo de coser; tempo de estar calado e tempo de falar;
Tempo de amar e tempo de aborrecer;
Tempo de guerra e tempo de paz."

(*Eclesiastes, 3:1-8*)

A Deus, que está no controle de tudo.

A Ricardo, que me transmite amor em todos os momentos.

A Daniela, que é pura ternura e carinho.

A Marcelo, que adoça os meus dias.

A Gabriel, que alegra a minha vida.

A Walfredo, que me fez ultrapassar meus limites.

Agradecimentos

Agradeço à Dataprev, na pessoa de Rômulo, que me proporcionou a oportunidade de chegar até aqui através do Programa de Incentivo à Pós-graduação. A Mauro, pela paciência e bom humor para entender minha ausência. A Júlio César por estar sempre disposto a esclarecer minhas dúvidas, mesmo as mais triviais. E também à Equipe de Operações, especialmente aqueles que sempre me fizeram sentir parte dela, ainda que distante fisicamente.

Agradeço a Aninha que facilita a vida de todos nós, alunos. Ao corpo de professores da COPIN, pela dedicação e empenho em nos tornar Mestres. Ao professor Fubica que me fez passar pela prova da peneira e perceber que podia ir adiante. Ao professor Antônio Berto que, no momento mais difícil, ampliou o meu conceito sobre verdade.

Agradeço aos meus filhos Daniela, Marcelo e Gabriel por aceitarem os momentos em que eu "vivia" no computador, embora sem compreender bem o porquê. Especialmente a Ricardo por nunca reclamar, por me incentivar, por acreditar em mim, por me amar muito e por ser um companheirão, sempre. Aos meus pais pela educação que me proporcionaram. À vovó, pela acolhida carinhosa nas minhas idas e vindas.

Finalmente, agradeço a Walfredo pela paciência de esperar que eu enxergasse as conclusões que ele já vira anos-luz adiante. Por me incentivar. Por nunca forçar demais. Por ser tão ocupado, mas ter sempre tempo disponível. Por ter o dom de ensinar todo o tempo, até no cafezinho. Por ser amigo. E por me fazer acreditar que não há limites, embora às vezes pareça impossível ir além.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo I

1. Introdução

O poder computacional oferecido pelo processamento paralelo está cada vez mais acessível aos usuários de computadores. Através das plataformas de processamento existentes hoje e de muitas outras em desenvolvimento, já é possível construir um supercomputador paralelo a partir de um conjunto de máquinas, a exemplo do que acontece com *Beowulf* [47]. Entretanto, se você não pode construir o seu próprio supercomputador, existe a alternativa de utilizar a computação em *Grid* [16], uma plataforma que provê meios para que um usuário possa utilizar recursos computacionais distribuídos pelo mundo.

Na verdade, para cada classe de aplicações paralelas existe uma plataforma que melhor se adequa às suas características. Entre as principais plataformas de processamento paralelo estão os supercomputadores paralelos de memória distribuída, também conhecidos por *MPPs* (Processadores Massivamente Paralelos). Estes supercomputadores são formados por um conjunto de nós (processador e memória) independentes, interconectados por redes dedicadas e muito rápidas [12]. Possuem a característica de ter um escalonador central que controla os programas a serem processados a partir de uma fila de execução. É sobre esta classe de supercomputadores que estaremos tratando neste trabalho, os quais vamos chamar de supercomputadores paralelos.

Os supercomputadores paralelos surgiram para atender a crescente demanda por capacidade de processamento. As aplicações se tornaram muito complexas e volumosas, conseqüentemente consumindo um tempo muito grande para a obtenção de resultados. Em um ambiente onde aplicações demoram dias, semanas e até anos para serem processadas, como é o caso do mapeamento do genoma humano ou das simulações climáticas da terra, a diferença de performance pode ser crucial. Para uma aplicação que é executada em uma hora, esta diferença de performance pode até não ser percebida. Porém, em um processamento que demora um ano, uma melhoria de performance de 50% antecipa o resultado em 6 meses.

A questão é: como melhorar a performance dos supercomputadores paralelos? A performance destes supercomputadores é afetada por diversos fatores característicos ao seu ambiente. Há o método de gerenciamento da fila de execução, conhecido como algoritmo de escalonamento, que procura otimizar a maneira que os programas serão alocados para processamento [25]. Outro fator, a carga de trabalho, conhecida como *workload*, possui características que influenciam diretamente o comportamento do sistema, tais como a distribuição dos tempos de execução e a qualidade dos pedidos de submissão dos programas que a compõem, entre outras [22] [36]. E, surpreendentemente, têm sido comprovado que a métrica utilizada para representar o desempenho do sistema também pode influenciar nos resultados da avaliação de performance [20] [21] [42]!

Com a finalidade de otimizar a performance das aplicações submetidas a estes supercomputadores, foi desenvolvido SA, um escalonador de aplicações. SA trabalha a partir de alternativas de submissão de um programa fornecidas pelo usuário, realizando uma projeção da execução e assim escolhendo a melhor alternativa de submissão. Estes programas que podem ser submetidos de diversas maneiras são conhecidos como programas moldáveis. SA efetivamente melhora a performance dos programas moldáveis [9].

O objetivo do nosso trabalho é estudar alguns aspectos das interações entre os fatores que afetam a performance dos supercomputadores paralelos. Com relação à métrica de performance, analisamos aquelas que são mais usadas atualmente e propomos uma métrica que interfira menos no processo de avaliação, que seja pouco tendenciosa e que traga mais informações sobre a relação sistema × *workload*. Esta métrica é chamada de Performance Relativa [41]. Com relação ao método de escalonamento, realizamos um estudo sobre o uso do escalonador de aplicação SA. SA tem o objetivo de melhorar a performance da aplicação que escalona, e efetivamente alcança este objetivo [9]. Mais do que isso, como comportamento agregado conseqüente do uso de múltiplas instâncias de SA, o próprio sistema tem sua performance melhorada em algumas situações [11]. SA escolhe a melhor alternativa de escalonamento no momento da submissão. Avaliamos a possibilidade de SA atuar novamente no momento do início da execução, com a expectativa de melhorar ainda mais a performance do programa. Tanto para a métrica como

para o método de escalonamento, realizamos experimentos e apresentamos os resultados obtidos.

1.1. Sumário da Dissertação

Nós mostramos nesta dissertação como a métrica de performance e o método de escalonamento podem influenciar no desempenho dos supercomputadores paralelos.

Apesar do resultado da avaliação de performance aparentemente depender apenas do escalonador e da carga de dados sendo processada, há casos na literatura que mostram a influência de outro fator: a métrica [20] [21] [25] [42]. Baseados nestes conflitos de resultados aparentemente gerados pelas métricas, realizamos um experimento prático que constata que a métrica utilizada pode alterar o resultado da avaliação de performance. A fim de esclarecer melhor esta questão, propomos uma métrica, **Performance Relativa**, que procura minimizar o efeito da métrica sobre a avaliação de performance [41]. Isto é possível porque Performance Relativa utiliza toda a distribuição dos resultados e não se baseia em estatísticas para a sumarização dos dados. Além disso, Performance Relativa tem o objetivo de prover informação que permita compreender melhor a atuação do escalonador sobre os dados processados pelo supercomputador.

O método de escalonamento é crucial para a performance dos supercomputadores paralelos, pois trata do modo que os programas são alocados da fila de espera para a execução. Programas moldáveis são aqueles que podem ser executados em partições de diversos tamanhos. Como a maioria dos programas paralelos hoje são moldáveis [14], esta característica pode ser aproveitada para melhorar a performance destes programas. SA é um escalonador de aplicação que tem o objetivo de escolher a melhor alternativa de escalonamento para um programa moldável [9]. SA atinge este objetivo e, além disso, também melhora a performance do sistema em certas situações, quando várias instâncias de SA estão sendo usadas no mesmo supercomputador [11]. A fim de investigarmos a possibilidade de melhorar ainda mais a performance alcançada pelo uso de SA, realizamos uma modificação na sua implementação. SA na sua forma original decide qual o tamanho da partição a ser usada no momento da submissão do programa. A nossa modificação acrescenta uma instância de SA dentro do escalonador de recursos do sistema, de forma

que SA possa ser invocado também no momento em que um programa é adiantado na fila de espera e pode iniciar sua execução imediatamente. Chamamos esta alteração de **SA no início da execução**. Verificamos que esta modificação melhora isoladamente a performance da aplicação que SA escalona. Entretanto, este comportamento piora o desempenho global do sistema quando muitas instâncias de SA estão sendo usadas. Os resultados mostram que não parece ser interessante para o sistema ter SA atuando tanto no momento da submissão como no momento do início da execução do programa. Confirmamos através das nossas simulações que o uso de SA apenas no momento da submissão, isto é, na sua forma original, é melhor para a performance do sistema como um todo, quando muitas instâncias de SA estão atuando. Detalharemos esta análise no capítulo V, Escalonamento e Moldabilidade.

1.2. Estrutura da Dissertação

Esta dissertação está organizada em seis capítulos. Este capítulo delineia a pesquisa e apresenta o cenário do nosso estudo, delimitando a sua área de abrangência. O segundo capítulo descreve os supercomputadores e a maneira como eles trabalham. O terceiro capítulo discute a avaliação de performance dos supercomputadores paralelos e os fatores envolvidos nesta questão. O quarto capítulo mostra a métrica de performance proposta, Performance Relativa, juntamente com o estudo de caso realizado sobre o assunto. O capítulo cinco apresenta uma discussão sobre o impacto do escalonador sobre a performance dos supercomputadores paralelos através do estudo de alternativas de escalonamento de programas moldáveis e o respectivo estudo de caso. Finalmente, o capítulo seis apresenta as conclusões do nosso trabalho.

Capítulo II

2. Supercomputadores

O crescimento da utilização de computadores para a resolução de problemas intensificou-se nas últimas décadas. Como os processadores têm-se tornado cada vez mais rápidos, seria possível supor que em algum momento não fosse mais necessário aumentar o seu poder computacional. Entretanto, historicamente, quando uma tecnologia em particular satisfaz as aplicações conhecidas, novas aplicações surgem demandando o desenvolvimento de novas tecnologias [28].

Mesmo com o crescimento acelerado da capacidade computacional dos processadores (praticamente dobrando a cada 18 meses [16]), existem hoje aplicações que demandariam muito tempo de processamento seqüencial. São aplicações intensivas em processamento e/ou intensivas em dados. Estas aplicações possuem cálculos extremamente complexos, demandando muito tempo para que sejam concluídas; outras, por sua vez, processam enorme quantidade de dados, consumindo dias, meses e até anos de execução.

A fim de possibilitar a obtenção de resultados de forma ainda mais rápida, surgiu o processamento paralelo. Tradicionalmente, o desenvolvimento da computação de alta performance foi motivado por computações numéricas de grande complexidade, como aplicações de previsão do tempo, projeto de dispositivos mecânicos e circuitos eletrônicos e simulações de reações químicas. Atualmente o que tem impulsionado a procura por computadores mais rápidos são tipicamente aplicações que necessitam do processamento de grande quantidade de dados, tais como gráficos avançados para tomografia computadorizada [48], realidade virtual, vídeo conferência, diagnósticos auxiliados por computador na medicina, simulações climáticas da terra, simulação do comportamento das galáxias e mapeamento genético [45], entre outras. Estas aplicações se beneficiam do processamento paralelo para acelerar a obtenção dos resultados.

2.1. Processamento Paralelo

Uma aplicação paralela é composta por várias tarefas. As tarefas que compõem uma aplicação paralela executam em vários processadores, caracterizando desta forma o paralelismo da aplicação e conseqüente redução no seu tempo de execução.

Os processadores usados por uma determinada aplicação constituem a *plataforma de execução* da aplicação. A compreensão das plataformas de execução de processamento paralelo é essencial para que se possa escolher e aproveitar as características de cada uma que favoreçam uma determinada classe de aplicações. Vejamos a descrição proposta por Cirne em [12], que classifica as plataformas de execução de aplicações paralelas de acordo com sua conectividade, heterogeneidade, compartilhamento, imagem do sistema e escala. Esta classificação baseia-se no escalonador.

Conectividade diz respeito aos canais de comunicação que interligam os processadores que compõem a plataforma de execução.

Heterogeneidade trata das diferenças entre os processadores, que podem ser de velocidade e/ou arquitetura.

Compartilhamento versa sobre a possibilidade dos recursos usados por uma aplicação serem compartilhados por outras aplicações.

Imagem do sistema se refere à existência de uma visão única da plataforma, independente do processador sendo utilizado. Por exemplo, todos os processadores da plataforma enxergam o mesmo sistema de arquivos?

Escala estabelece quantos processadores tem a plataforma.

Cada aplicação paralela tem uma série de requisitos, que podem ser melhor ou pior atendidos por uma dada plataforma. Em princípio, procuramos executar uma aplicação paralela em uma plataforma adequada às características da aplicação. Por exemplo, considere conectividade, um aspecto em que plataformas diferem consideravelmente. Para obter boa performance de uma aplicação paralela cujas tarefas se comunicam e sincronizam freqüentemente, necessitamos utilizar uma plataforma de execução com boa conectividade.

Podemos agrupar as plataformas de execução hoje existentes em quatro grandes grupos: *SMPs*, *MPPs*, *NOWs* e *Grids*.

SMPs (ou multiprocessadores simétricos) são máquinas em que vários processadores compartilham a mesma memória. Multiprocessadores simétricos possibilitam um fortíssimo acoplamento entre os processadores e executam uma única cópia do sistema operacional para todos os processadores. Portanto, eles apresentam uma imagem única do sistema e excelente conectividade. Todavia, multiprocessadores simétricos apresentam limitações em escalabilidade, raramente ultrapassando 16 processadores. Multiprocessadores são relativamente comuns no mercado e vão desde máquinas biprocessadas *Intel* até grandes servidores como os *IBM pSeries*. A Figura 1 mostra a arquitetura de um multiprocessador.

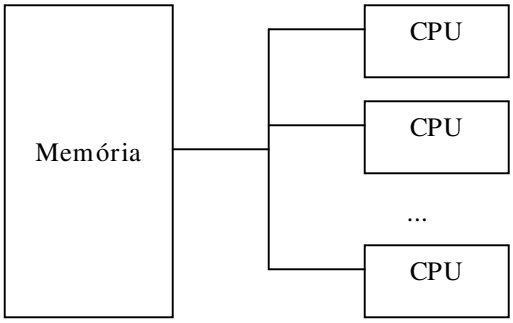

Figura 1 – Multiprocessador [12]

MPPs (ou supercomputadores paralelos) são compostos por vários nós (processador e memória) independentes, interconectados por redes dedicadas e muito rápidas. *MPPs* incluem supercomputadores paralelos como o *IBM SP2* e *Cray T3E*, como também *clusters* de menor porte montados pelo próprio usuário. Tipicamente cada nó roda sua própria cópia do sistema operacional, mas uma imagem comum do sistema é implementada através da visibilidade dos mesmos sistemas de arquivo por todos os nós. Um *MPP* é controlado por um escalonador que determina quais aplicações executarão em quais nós. Ou seja, não se pode utilizar um nó que não tenha sido alocado à aplicação pelo escalonador. A Figura 2 apresenta a arquitetura de um *MPP*. *MPPs* são a plataforma alvo deste trabalho.

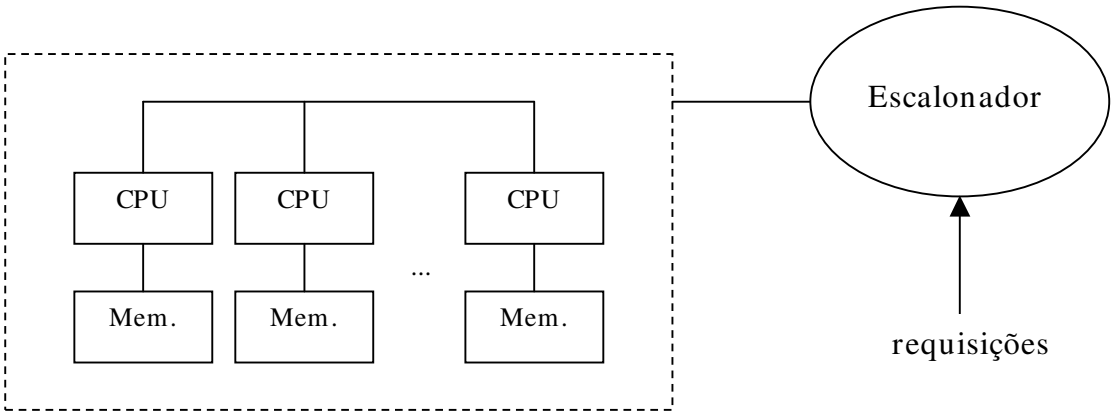

Figura 2 – *MPP* [12]

NOWs (ou redes de estações de trabalho) são simplesmente um conjunto de estações de trabalho ou *PCs*, ligados por uma rede local [1]. *NOWs* são arquiteturalmente semelhantes aos *MPPs*. Ambas plataformas são formadas por nós que agregam processador e memória. Uma diferença entre *NOWs* e *MPPs* é que os nós que compõem um *MPP* tipicamente são conectados por redes mais rápidas que as que conectam os nós das *NOWs*. Mas a principal diferença entre ambas arquiteturas é que *NOWs* não são escalonadas de forma centralizada. Isto é, em uma *NOW*, não há um escalonador para o sistema como um todo. Cada nó tem seu próprio escalonador local. Como resultado, não há como dedicar uma partição da *NOW* a uma só aplicação paralela. Portanto, uma aplicação que executa sobre a *NOW* deve considerar o impacto sobre sua performance da concorrência por recursos por parte de outras aplicações. A Figura 3 ilustra esquematicamente uma *NOW*.

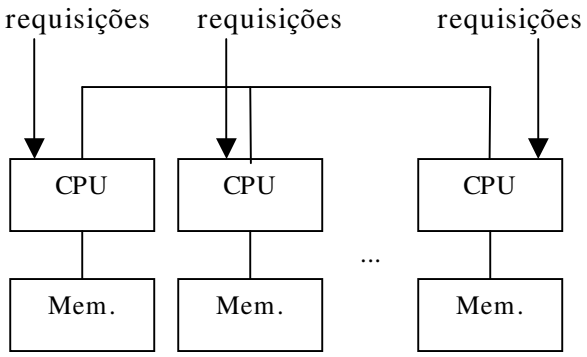

Figura 3 - *NOW[12]*

Grids são o passo natural depois das *NOWs* no sentido de mais heterogeneidade e maior distribuição. Os componentes de um *Grid* não se restringem a processadores, podendo ser *SMPs* e *MPPs* como também instrumentos digitais. *Grids* tipicamente não fornecem uma imagem comum do sistema para seus usuários. Componentes do *Grid* podem variar drasticamente em capacidade, software instalado, sistemas de arquivo montados e periféricos instalados. Além disso, os componentes de um *Grid* podem estar sob o controle de diferentes entidades e, portanto, em domínios administrativos diversos.

Conseqüentemente, um dado usuário pode ter acesso e permissões bastante distintas nos diferentes componentes de um *Grid*. Naturalmente, o *Grid* não pode ser dedicado a um usuário, embora seja possível que algum componente possa ser dedicado (um *MPP*, por exemplo). A Figura 4 mostra um *Grid* onde temos um microscópio fazendo instrumentação eletrônica e um computador armazenando grande quantidade de dados.

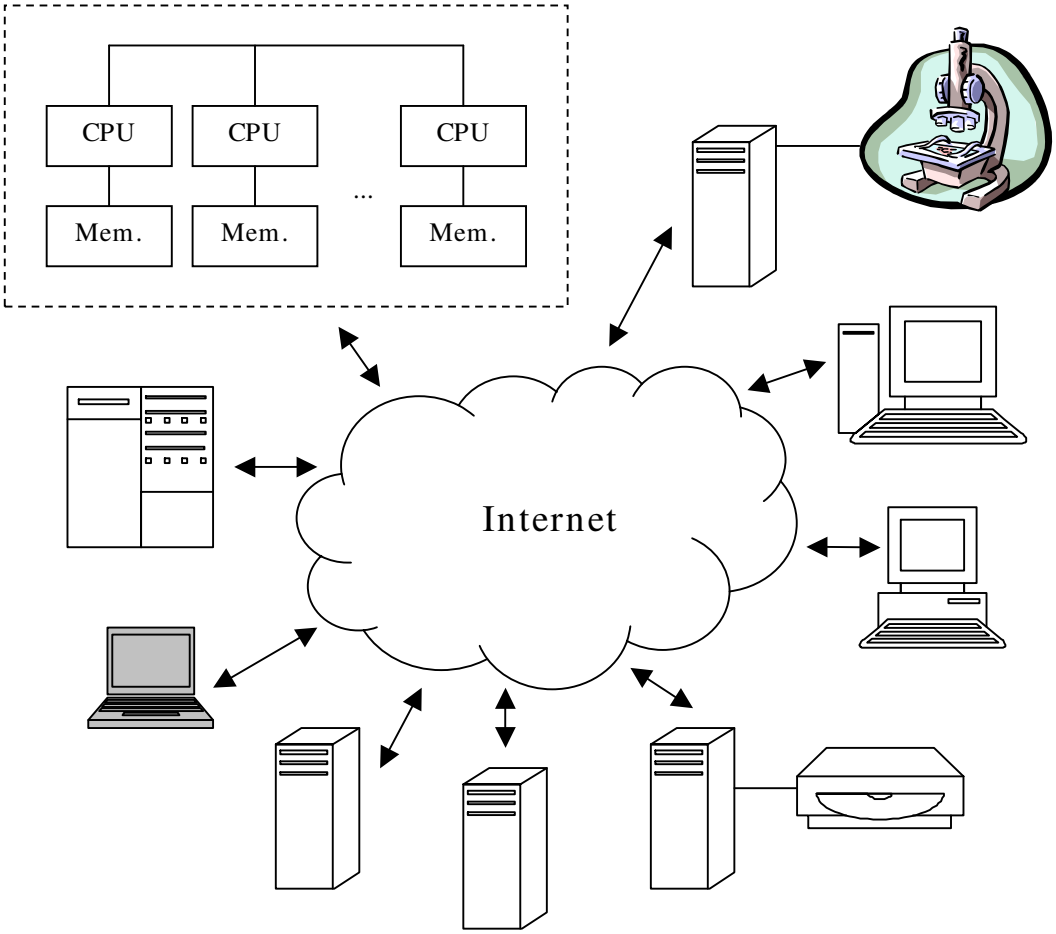

Figura 4 - *Grid* Computacional [12]

Como já dissemos, o domínio do nosso trabalho está inserido na área dos *MPPs*, ou Supercomputadores Paralelos de Memória Distribuída, chamados aqui de supercomputadores paralelos.

2.2. Supercomputadores Paralelos

Um supercomputador paralelo é composto de um certo número de processadores trabalhando cooperativamente, cada um com a sua própria memória, conectados por uma rede interna extremamente rápida. Os programas executados neste ambiente tipicamente possuem tarefas que se comunicam e sincronizam.

Neste ambiente paralelo um programa deve explicitar o grau de paralelismo, isto é, o número de processadores ou *nodes* (*n*) - também conhecido como *tamanho da partição* ou simplesmente *tamanho* - e o tempo requisitado (*tr*) necessários à sua execução [9]. A relação entre o número de processadores e o tempo requisitado é chamada de *forma* ou *área*, como mostrado na Figura 5.

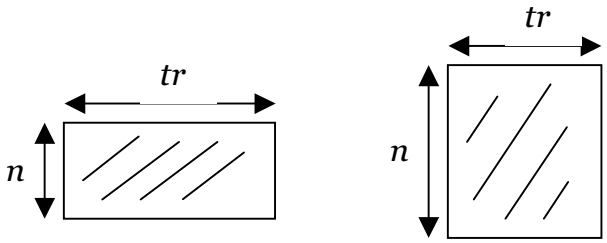

Figura 5 - Forma ou área de um programa

Um programa submetido ao supercomputador pode não encontrar recursos disponíveis para ser executado imediatamente e então é colocado em uma *fila,* a qual é controlada pelo *escalonador* do supercomputador.

O escalonador tem a função de receber as requisições dos programas e decidir quando estes iniciarão a execução e que processadores usarão. Ou seja, o escalonador decide qual programa da fila de execução será o próximo a ser executado. O escalonamento nos supercomputadores paralelos é um tipo de escalonamento *on-line*, pois lida com programas que chegam continuamente ao sistema [17].

Capítulo III

3. Performance

Avaliação de Performance é difícil. Em ambientes paralelos, em particular, o processo de avaliação de performance é mais crucial e complexo do que em ambiente seqüenciais [38]. A performance dos sistemas paralelos é influenciada por uma multiplicidade de fatores relacionados à estrutura das aplicações e à sua habilidade em explorar as características de paralelismo do sistema [7], além de sofrer influência do modo como o sistema paralelo trabalha. Não vamos explorar aqui as características das aplicações, o que seria um outro campo de pesquisa, mas sim o ambiente onde estas aplicações são executadas e os fatores que influenciam na avaliação de performance destes sistemas computacionais.

3.1. Medindo Performance

A avaliação de performance é freqüentemente realizada com o objetivo de comparar diferentes implementações de sistemas. Faz parte da avaliação trazer à tona diferentes aspectos de desempenho observados durante a comparação. Estas diferenças de desempenho refletem as características próprias e individuais de cada sistema sendo avaliado. Entretanto, as diferenças de performance encontradas podem estar sendo influenciadas pela metodologia utilizada durante a avaliação, pois a performance de um sistema não é apenas função de seu projeto ou de sua implementação, mas pode ser afetada pela carga à qual o sistema está submetido, e ainda pela métrica que está sendo utilizada para representar o desempenho observado [24].

A técnica mais comumente utilizada para avaliar a performance de supercomputadores é o *benchmarking,* que consiste em submeter o sistema a um conjunto representativo e abrangente de carga a fim de analisar o seu comportamento [7]. Existem ferramentas de software que se propõem a auxiliar o avaliador neste processo, oferecendo uma massa de dados projetada de forma a representar amplamente as situações reais, além de fornecer como saída gráficos e relatórios que

serão interpretados pelo avaliador de performance. Estes *toolkits* têm o objetivo de automatizar o processo de avaliação, diminuir o tempo necessário para efetivação dos experimentos e minimizar o número de erros na avaliação, em suma aumentando a produtividade do processo [7]. Porém, usando uma ferramenta como esta, o avaliador deverá ter um pouco menos de trabalho, mas não poderá deixar de analisar se os dados são mesmo representativos, se a métrica usada foi adequada e se os resultados estão bem representados, entre outros aspectos a considerar. Em resumo, o *toolkit* poderá ajudar o avaliador, mas não suprime a etapa de análise minuciosa de todos os aspectos que envolvem uma avaliação de performance.

Jain propõe em [33] uma abordagem sistemática do estudo de medição de performance :

- Definir o sistema e os objetivos que este deseja alcançar;
- Descrever o que o sistema faz e quais são as respostas a serem consideradas;
- Selecionar as métricas de avaliação a serem utilizadas;
- Listar os parâmetros considerados;
- Selecionar os fatores a serem estudados, isto é, os parâmetros que irão variar para produzir os diferentes resultados;
- Selecionar as técnicas de avaliação, como por exemplo simulação, medição, análise.
- Selecionar a carga (*workload*) a ser aplicada para geração dos resultados;
- Projetar os experimentos;
- Analisar e interpretar os dados obtidos;
- Apresentar os resultados.

Para cada observador pode haver um conjunto de métricas significativas, dependendo do que se deseja avaliar no sistema computacional. Por exemplo, a análise de desempenho de um programa pode ser feita em função da quantidade de tempo de CPU utilizado, do tempo total de E/S (entrada/saída) ou da utilização de memória, dependendo de qual componente do sistema se está analisando: CPU, subsistema de E/S ou subsistema de memória virtual.

Para os supercomputadores paralelos, vamos considerar performance através do ponto de vista de um usuário que submete o seu programa e espera obter uma resposta *"o mais rápido possível"*. O problema é que embora "*o mais rápido*

possível" seja não-ambíguo para uma determinada aplicação, métricas que sumarizam os resultados para toda a *workload* podem ser enganadoras. Há exemplos na literatura que mostram a métrica influenciando nos resultados da avaliação [20] [21] [25] [42]. Estes exemplos mostram casos onde, usando métricas diferentes para avaliar o desempenho de escalonadores, foram obtidos resultados contraditórios.

Em princípio a performance deveria depender apenas da *workload* que está sendo processada [17] [36] e do método de escalonamento utilizado para gerenciar a fila de execução dos programas [23]. Hoje já se conhece que a realidade não é bem assim. Há uma interação entre os fatores que estão presentes no contexto dos supercomputadores paralelos: Escalonador, *Workload* e Métrica [20]. Vamos examinar aqui cada um destes aspectos isoladamente para em seguida nos aprofundarmos na influência da métrica e do escalonador. Esta escolha nos pareceu mais interessante, desde que a influência da *workload* é hoje amplamente investigada na literatura [17] [18] [22] [36] [37] [39] [53].

Além de todos estes fatores, é preciso ainda tomar cuidado ao representar os resultados obtidos, pois gráficos com escalas pouco claras ou mal representadas, comparações entre equipamentos de configurações diferentes ou mesmo de características diferentes e o uso de unidades de medida inadequadas podem levar a resultados distorcidos e enganosos [3].

3.2. *Workload*

A performance de um sistema paralelo depende em parte da carga que ele está processando [7] [21] [36]0. Deste modo, para comparar e avaliar diferentes sistemas computacionais paralelos é necessário caracterizar a carga à qual serão submetidos.

Existem duas maneiras mais comuns de usar uma carga de dados para avaliar um sistema [17]:

- Utilizar uma amostra de dados real, coletada em um período de tempo *(workload trace* ou *log* de sistema*)*, ou
- Criar um modelo de carga *(workload model)* a partir de dados observados anteriormente e usar o modelo para fazer análises e simulações.

A vantagem de usar um *trace* é que torna o teste mais realista, pois os dados refletem precisamente a carga real com toda a sua complexidade, mesmo que esta complexidade não seja conhecida por quem realiza o teste. A desvantagem é que o *trace* reflete uma carga específica, e é sempre questionável se o resultado pode ser generalizado para outros sistemas.

Outro aspecto com relação ao *trace* é que ele reflete a carga aplicada a uma determinada configuração, portanto o mesmo *trace* quando usado como modelo de carga para outra configuração torna difícil a comparação entre os dois sistemas [17].

A característica mais relevante do modelo de carga sintético é que ele contém exatamente o que o modelador deseja. Isto pode ser uma vantagem ou uma desvantagem: vantagem porque o modelador conhece as características do modelo e pode controlá-las, e desvantagem porque os *traces* podem conter certas características que são desconhecidas do modelador, e portanto não serão incluídas no modelo de carga construído [8].

Algumas outras vantagens que os modelos de cargas sintéticos possuem sobre os *traces* são:

- Quem modela tem amplo conhecimento da característica da carga, sendo fácil determinar quais parâmetros estão relacionados com os outros, e esta informação é parte integrante do modelo;
- É possível alterar os parâmetros do modelo a fim de investigar a sua influência sobre o resultado. Isto permite medir a sensibilidade do sistema a diferentes parâmetros. Também é possível selecionar os parâmetros do modelo que mais se adequam à avaliação de um determinado computador, ou conjunto de computadores sendo avaliados;
- Um modelo não é afetado pela política de um determinado local, vigente na época em que o *trace* foi gravado: por exemplo, pode existir uma política de não permitir que programas com tempo de execução superior a quatro horas sejam executados, o que modificará a distribuição natural de execução;
- Os *logs* podem estar "poluídos" por dados inválidos, como a inclusão de programas que foram cancelados porque excederam os limites de uso de

recursos. Em um modelo sintético, tais programas podem ser evitados ou modelados explicitamente, se desejado;

- Finalmente, a modelagem pode aumentar o nosso entendimento e levar a novos projetos baseados neste conhecimento. Por exemplo, é possível projetar uma política de gerenciamento de recursos parametrizada por um modelo de carga. À medida que os *logs* apresentarem características diversas do modelo, novas políticas podem ser definidas, assim como o modelo pode sofrer transformações.

Vale ressaltar que o estudo de modelagem de carga vem sendo influenciado pelas novas características de sistemas paralelos. A metodologia geralmente usada em ambientes de processamento seqüencial nem sempre provê uma descrição correta de carga em arquiteturas paralelas [39].

O principal problema com modelos de carga sintéticos, assim como *traces*, é a sua representatividade. Isto é, com que grau de semelhança o modelo representa a carga que o sistema irá encontrar na prática. Será que o comportamento do sistema quando do uso do modelo de carga representa o seu comportamento quando em operação real? O modelo contempla todas as situações que o sistema irá encontrar na prática?

3.3. Escalonadores

O método de escalonamento trata de como os programas que estão na fila de espera serão alocados para execução. Na prática, o comportamento dos escalonadores de supercomputadores variam de máquina para máquina, e até quando o mesmo software de escalonamento é usado, cada local estabelece sua própria política de escalonamento, o que implica na mudança de comportamento dos escalonadores [9]. Assim, dependendo da política de escalonamento escolhida, os sistemas podem apresentar variações de desempenho, influenciando na performance do supercomputador paralelo.

Como já vimos, em um ambiente de processamento paralelo um programa deve explicitar o número de processadores (n) e o tempo requisitado (tr) para a sua execução. Note que o programa é abortado pelo sistema caso ele exceda o tempo requisitado.

Um programa submetido pode não encontrar recursos disponíveis para ser executado imediatamente. Quando isso acontece, o programa é colocado em uma fila, a qual é controlada pelo escalonador do supercomputador. O escalonador tem a função de receber as requisições dos programas e decidir quando estes iniciarão a sua execução e que processadores usarão. O algoritmo usado para alocar os programas é conhecido como método de escalonamento. Obviamente, dependendo da política de escalonamento escolhida, os sistemas podem apresentar variações de desempenho.

A prática atual no escalonamento de programa de supercomputadores paralelos apresenta algumas características comuns [9]:

- Os programas podem começar fora de ordem (isto é, mesmo que o programa x chegue antes de y, y pode começar antes de x);
- Os recursos ociosos são reutilizados (isto é, o tempo que foi requisitado e não foi usado pode ser alocado para outros programas);
- A prioridade é baseada no tempo (um programa que chegou há mais tempo tem maior chance de ser o próximo a executar do que um que chegou há pouco tempo) e
- Existe algum mecanismo para evitar que os programas grandes demorem muito para executar.

Apesar de existirem vários escalonadores atualmente (Easy [35], PBS [32], Maui [40], LSF [46], CRONO [43], entre outros) vamos descrever os métodos de escalonamento *Conservative backfilling* e *Easy backfilling*. Estes métodos implementam os quatro mecanismos descritos acima e serão objetos do nosso estudo de caso. São duas variações do método *backfilling*. *Backfilling* é uma otimização do método FCFS (*First Come First Served*), onde os programas que chegam para executar são colocados em uma fila e alocados por ordem de chegada. Como os programas requerem um determinado número de processadores para executar (n), pode acontecer de haver processadores disponíveis, mas não em quantidade suficiente para que o primeiro programa na fila possa executar. *Backfilling* tenta reduzir ao mínimo esta subutilização de recursos através da alocação de outros programas da fila capazes de preenchê-los utilizar os processadores disponíveis [42].

Conservative Backfilling

No algoritmo do *Conservative backfilling*, quando o escalonador encontra o primeiro programa que não dispõe de recursos para ser executado imediatamente, percorre a fila procurando um outro que possa ser executado com os recursos que estão disponíveis no momento. Este programa pode então ser adiantado, desde que não atrase o início estimado de nenhum outro ocupante da fila de espera.

Deste modo, o algoritmo garante previsibilidade, pois no momento da submissão já é possível garantir um limite máximo para o início de um programa (máximo porque os programas à sua frente na fila poderão terminar antes ou então o próprio programa poderá ser adiantado através de *backfilling*).

Conservative backfilling usa uma lista de alocação que informa, em qualquer momento, quais processadores estão sendo utilizados por quais programas [25]. Os programas que chegam ao sistema são processados usando *FCFS*. Por exemplo, a Figura 6 mostra a submissão de cinco *requests* na seguinte ordem: A, B, C, D e E. Perceba que C é colocado antes de B porque no momento em que *time=1* os recursos disponíveis não são suficientes para que B seja executado, mas atendem a C.

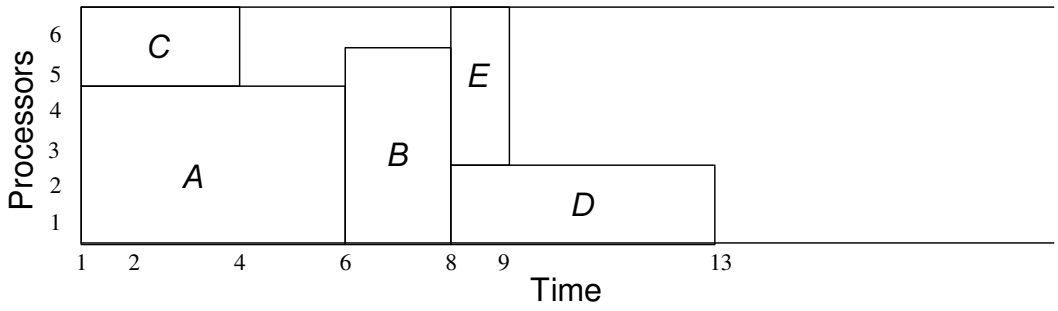

Figura 6 - Lista de alocação após a submissão de cinco *requests* [9]

Quando um programa termina usando menos tempo do que foi requerido, *Conservative backfiling* examina a fila pela ordem de submissão e adianta o primeiro programa que cabe no *slot* que acabou de tornar-se vago. Se este remanejamento criar outro *slot*, este é preenchido da mesma forma. O processo termina quando não há mais *backfilling* a ser feito. A Figura 7 mostra o que acontece quando A termina no *time=2*: B é promovido para iniciar logo depois de C, D segue B,

mas E pode terminar antes do início estimado de B e então é adiantado para o início da fila, de modo que possa começar sua execução imediatamente.

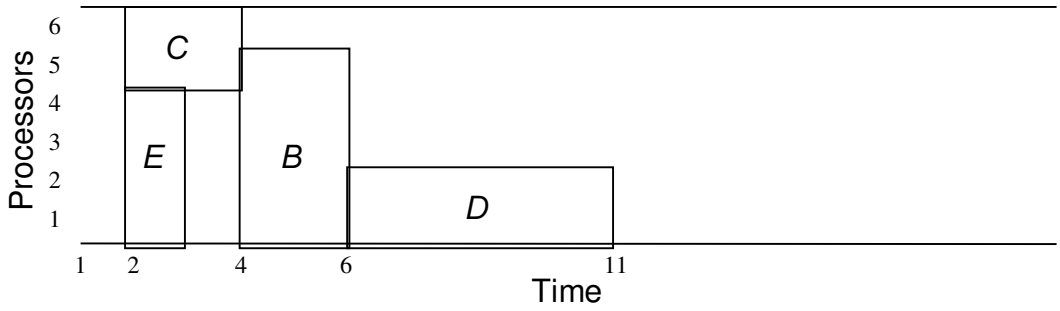

Figura 7 - Lista de alocação após o *backfilling* iniciado por A terminar no *time=2* [9]

Easy Backfilling

O algoritmo de *Easy backfilling* também percorre a fila no momento em que um programa não pode ser executado, a fim de encaixar um outro que aproveite os recursos disponíveis. A diferença é que este método é mais agressivo, pois permite adiantar qualquer programa da fila de espera desde que isto não cause atraso ao primeiro da fila [42].

Este algoritmo ganha em agressividade, propondo uma maior utilização do sistema, mas perde em previsibilidade, pois é impossível estabelecer qualquer garantia de quando um programa será executado. Por exemplo, a Figura 8 mostra a submissão de cinco *requests* nesta ordem: A, B, C, D e E.

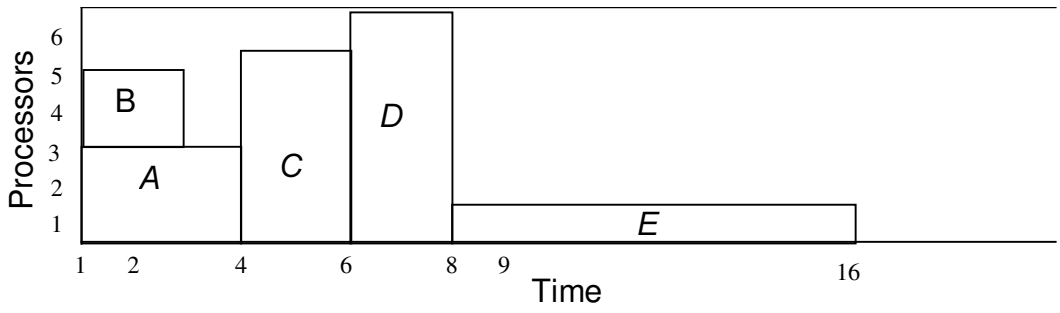

Figura 8 - Lista de alocação antes do *Easy backfilling*

Quando um programa termina usando menos tempo do que o requisitado, *Easy backfilling* percorre a fila e adianta o primeiro programa que couber no *slot* que ficou vago. A diferença é que os outros programas da fila podem ter seu início estimado postergado, exceto o primeiro. Vejamos quando A termina no *time=2*: B

já está executando, C é o primeiro da fila e E pode ser adiantado pois há recursos suficientes para atendê-lo sem atrasar C, embora isto provoque o atraso do início estimado de D. D tinha início previsto no *time=6* e após o *backfilling* sua nova previsão é iniciar no *time=10*. A Figura 9 mostra a lista de alocação após o *Easy backfilling*.

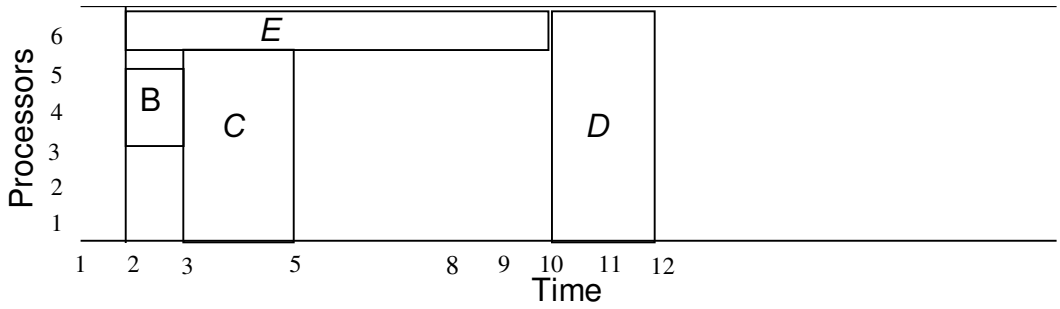

Figura 9 - Lista de alocação após o *Easy backfilling*

3.4. Métricas

As métricas são os métodos utilizados para representar o desempenho de um sistema. Existem algumas métricas comumente utilizadas na avaliação de performance de supercomputadores paralelos, sendo que cada uma delas possui características diferentes que podem ser interessantes para um determinado tipo de sistema, mas pouco adequadas a outros.

A situação ideal é que a métrica não interfira na avaliação, mas na verdade a métrica pode ter um efeito determinante sobre a conclusão da avaliação de performance. Em [20] foi realizado um estudo que verifica a interação das métricas de performance com a *workload* e os métodos de escalonamento, e percebeu-se que estes fatores interagem e podem distorcer o resultado da avaliação. Nosso interesse é detalhar cada tipo de métrica a fim de conhecer suas características e procurar compreender como as métricas são capazes de favorecer um determinado tipo de escalonador ou *workload* e assim influenciar o resultado da avaliação de performance do supercomputador paralelo.

Turn-around time (*tt*) médio

O *turn-around time*, também conhecido como tempo de resposta, é definido como o intervalo entre o instante em que o programa é submetido ao sistema até o

momento em que toda a sua saída é produzida e o programa termina. O *turn-around time* é muito bom para medir a performance de uma aplicação. No entanto, queremos uma métrica que represente a performance do sistema, assim o *turn-around time* médio passa a ser uma escolha natural.

No entanto, quando usamos média aritmética, temos que considerar as características desta estatística sumarizadora. A média aritmética representa o "centro de gravidade" de um conjunto de observações [17]: $\bar{x} = \frac{1}{n}\sum x_i$. A média aritmética é a idéia que ocorre à maioria das pessoas quando se fala em "média", e possui certas propriedades interessantes e úteis que explicam porque é ela a medida de tendência central mais utilizada[52]:

- A média de um conjunto de números pode sempre ser calculada.
- Para um dado conjunto de números, a média é única.
- A média é sensível a (ou afetada por) todos os valores do conjunto. Assim, se um valor se modifica, a média também se modifica.

Entretanto, a média possui ainda a característica de ser influenciada por valores extremos. O problema é usar a média quando a distribuição é muito elástica. Vejamos um exemplo :

número de programas	***tt* (segundos)**	**média aritmética (segundos)**
100	1	1
99 1	1 2000	20.99

Tabela 1 - Média aritmética do *turn-around time*

Como vimos, os valores extremos influenciam bastante nesta medida, e apenas um valor mais alto pode elevar a média. (Neste caso a moda - valor que mais ocorre no conjunto - representaria melhor os dados, porém só é possível chegar a esta conclusão porque conhecemos os dados).

Problema relacionado à performance: Se a distribuição dos dados para avaliação do sistema for muito elástica, ou se não for conhecida, a performance calculada através desta métrica pode estar distorcida [22].

Slowdown médio

O *slowdown*, também conhecido como fator de expansão, é o *turn-around time* normalizado pelo tempo de execução: $s = tt / te$, onde tt é o *turn-around time* e te é o tempo de execução [23]. Lembre que $tt = tw + te$.

Assim, se o tempo de espera de um programa for igual ao seu tempo de execução, ele sofrerá de um fator de expansão de 2.

tempo de execução (*te*)	tempo de espera (*tw*)	*slowdown* (s = (*tw* + *te*)/*te*)
20 ms	20 ms	(20 + 20)/ 20 = 2

Tabela 2 - Cálculo do *slowdown*

O *slowdown* é utilizado a fim de reduzir valores extremos associados a programas muito longos. A idéia principal por trás do *slowdown* é que um programa espere na fila um tempo proporcional ao seu tempo de execução.

Problema relacionado à performance: O *slowdown* supervaloriza a importância dos programas muito curtos. Veja a Tabela 3, por exemplo. Um programa que é executado durante 100 ms e espera na fila por 10 minutos (600.000 ms) possui um *slowdown* de 6001. Já um programa que é executado por 10 segundos (10.000 ms) e espera na fila os mesmos 10 minutos (600.000 ms) possui um *slowdown* de 61. Observa-se que, em sistemas nos quais o número de processadores pode ser escolhido automaticamente pelo escalonador através de algumas alternativas possíveis de *requests*, o próprio sistema poderia melhorar o seu *slowdown* escolhendo utilizar menos processadores. Além disso, a utilização desta métrica pode encorajar o sistema a fazer com que os programas tenham um tempo de execução maior, para poder baixar o valor do *slowdown*.

tempo de execução (*te*)	tempo de espera (*tw*)	*slowdown* (s = (*tw* + *te*)/*te*)
100 ms	600.000 ms	(600.000 + 100)/ 100 = 6001
10.000 ms	600.000 ms	(600.000 + 10.000) / 10.000 = 61

Tabela 3 - *Slowdown* de programas muito curtos

Bounded slowdown médio

O *bounded slowdown* procura atenuar o *slowdown* através da definição de um limite para o tempo de execução, $bs = tt / max(te, \tau)$. A diferença é que, para programas muito pequenos, o *slowdown* será limitado pelo valor-limite τ escolhido, e não pelo tempo de execução. O comportamento desta métrica vai depender do valor de τ [23].

Problema relacionado à performance: Apresenta os mesmos problemas que slowdown. Além disso, como estimar o valor ideal de τ?

Per-processor bounded slowdown médio

Percebeu-se que programas que fazem a mesma quantidade de processamento e têm o mesmo *turn-around time* podem ter *slowdown* diferentes, se variarem a quantidade de processadores e o tempo, ou seja, se variarem a sua "forma".

Por exemplo, um programa que executa imediatamente em um único processador por 100 segundos tem um *slowdown* de 1, enquanto que um programa que espera 90 segundos na fila e então é executado em 10 processadores por 10 segundos tem um *slowdown* de 10. Os dois programas têm o mesmo tempo de processamento total e o mesmo *tt*, mas os valores de *slowdown* são diferentes. Esta verificação levou a proposição de uma nova métrica, denominada "*per-processor bounded slowdown*" [53]:

***pp-slowdown* = *tt* / *n* * max(*te*, τ),** onde *n* é o número de processadores usado pelo programa. Ou seja, divide-se o valor original do *bounded slowdown* pelo número de processadores utilizados. No exemplo, o valor do *pp-slowdown* para o programa que usa 10 processadores é normalizado para 1, enquanto que o programa que usa só um processador fica com o *pp-slowdown* também de 1.

Problema relacionado à performance : Apresenta os mesmos problemas que *slowdown* e *bounded slowdown*.

Média geométrica do *turn-around time*

A média geométrica é definida por $medgeom(x_1,...,x_n) = \sqrt[n]{x_1 * ... * x_n}$. Ela tem a característica de ser menos influenciada por valores extremos [9]. A idéia de usar a média geométrica ao invés da média aritmética tem o objetivo de reduzir o efeito dos programas com tempo de execução muito longo. Por esta razão, a média geométrica é usada para agregar o tempo de execução dos programas que compõem o *Spec Benchmark* [51].

Relação com a análise de performance: A média geométrica dos *turn-around times* é uma métrica utilizada para representar o desempenho do escalonador. Tem a propriedade de refletir igualmente uma melhoria de performance em qualquer um dos programas que compõem a *workload*. Por exemplo: seja P a performance de uma *workload* calculada através da média geométrica, $P = \sqrt[n]{x_1 * ... * x_n}$. Vamos chamar de P1 a nova performance obtida quando um dos programas da *workload,* digamos x_i , apresenta uma melhoria de 50% no seu *turn-around time:* $P1 = \sqrt[n]{x_1 * ... * \frac{x_i}{2} * ... * x_n}$. Então, $P1 = \frac{\sqrt[n]{x_1 * ... * x_n}}{\sqrt[n]{2}}$. Assim vemos que $P1 = \frac{P}{\sqrt[n]{2}}$, mostrando que a performance original foi melhorada em $\sqrt[n]{2}$, independentemente do *tt* de x_i . Qualquer que fosse o x_i melhorado em 50%, a média geométrica refletiria essa melhoria da mesma forma. Note que isso acontece porque, pela definição de média geométrica, $medgeom(\frac{x_1}{y_1},...,\frac{x_n}{y_n}) = \frac{medgeom(x_1,...,x_n)}{medgeom(y_1,...,y_n)}$.

Comparando com a média aritmética, a média geométrica é menos influenciada por valores extremos. Como tipicamente as *workload* possuem valores muito elásticos, o resultado da performance que envolve estas *workloads* sofrerá menos influência destes valores extremos se calculado através da média geométrica do que através da média aritmética. Além disso, a média geométrica reflete igualmente um incremento (ou diminuição) na performance de qualquer um dos programas que compõem a *workload*.

Problema: Podemos ter que $\sum x_i < \sum y_i$ e $medgeom(x_i) > medgeom(y_i)$. O somatório dá uma noção intuitiva de performance, pois representa o total do tempo de processamento. A relação acima pode acontecer se houver algum y_i muito grande, por exemplo, pois a média geométrica não é tão afetada por um único valor quanto o somatório.

Throughput

Podemos definir *throughput* como a quantidade de tarefas executadas em um certo período de tempo. Para sistemas paralelos, poderíamos pensar como a quantidade de programas processados em um certo período de tempo.

Problema: Esta métrica faz mais sentido para um sistema *on-line* fechado, onde assume-se que há um conjunto fixo de programas a serem processados. Para um sistema *on-line* aberto, onde novos programas chegam a todo momento, o *throughput* irá variar em função das características dos programas que estão chegando. Isto é, dependendo das características da *workload*, o *throughput* poderá variar consideravelmente.

Utilização

Utilização representa a razão entre a efetiva utilização do sistema e a sua capacidade total de processamento.

Problema: Em programas rígidos, a utilização é determinada pela maneira que os programas estão chegando para serem processados (*arrival rate)*, sendo razoável representar o desempenho do sistema em função da utilização. Já para programas moldáveis, mudar o tamanho da partição provavelmente irá alterar as características de execução do programa, afetando assim a utilização.

Capítulo IV

4. Performance Relativa

Acreditamos que boa parte do problema da interferência da métrica sobre o processo de avaliação de performance dos supercomputadores paralelos seja devido à natureza sumarizadora das métricas baseadas em estatísticas utilizadas até então. A sumarização oferecida por uma estatística perde muita informação, uma vez que as distribuições dos *turn-around times* são tipicamente muito elásticas nas *workloads* dos supercomputadores [36]0. Propomos como solução o uso da **Performance Relativa**, que se baseia em toda a distribuição dos *turn-around times* obtidos pelos escalonadores avaliados.

A análise de performance baseada em Performance Relativa parece sofrer menos interferência da *workload* do que as métricas baseadas em estatísticas sumarizadoras, uma vez que seu resultado inclui a análise de toda a distribuição e não usa sumarização dos dados. Possibilita também uma boa visualização gráfica comparativa dos resultados.

4.1. Definição

A performance relativa toma como base a relação entre os dois valores de *turn-around time* obtidos para um mesmo programa através das duas alternativas de escalonamento que se deseja comparar. A metodologia utilizada é a seguinte:

- Define-se quais escalonadores (ou supercomputadores) deseja-se comparar, dois a dois (chamemos de **a** e **b**);
- Toma-se uma determinada *workload* de tamanho **m** a ser submetida aos dois escalonadores **a** e **b**;
- Registra-se o *turn-around time* de cada programa obtido em cada escalonador (tt_a e tt_b);
- Calcula-se a razão entre os **m** pares dos *turn-around times*, (tt_a / tt_b);
- Finalmente, calcula-se a distribuição acumulada de freqüência das razões obtidas, a fim de obter a representação gráfica.

A Figura 10 mostra um exemplo de Performance Relativa.

Para cada razão **i** interpreta-se o resultado da seguinte maneira:

- Se $tt_a / tt_b = 1$, o programa obteve o mesmo *turn-around time* nos dois escalonadores;
- Se $tt_a / tt_b, < 1$ o programa obteve melhor *turn-around time* no escalonador **a**;
- Se $tt_a / tt_b, > 1$ o programa obteve melhor *turn-around time* no escalonador **b**.

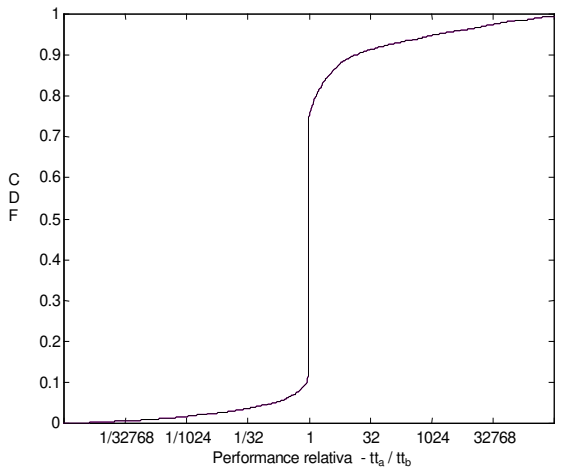

Figura 10 – Exemplo de Performance Relativa

Através da Figura 10 percebemos que aproximadamente 65% dos programas obtiveram o mesmo valor de *turn-around time* nos dois escalonadores; o escalonador **a** foi melhor em aproximadamente 11% dos *turn-around times* e o escalonador **b** foi melhor em aproximadamente 24% dos *turn-around times*. Mais ainda, a partir do gráfico é possível avaliar o impacto que os escalonadores **a** e **b** tiveram sobre os programas que beneficiaram. Por exemplo, alguns programas chegaram a concluir em tempo 32.000 vezes menor quando usando o escalonador **a**. Por outro lado, o escalonador **b** concluiu alguns programas em tempo 1.000.000 de vezes menor que o escalonador **a**.

O uso desta métrica é interessante porque analisa toda a distribuição da *workload*, ao contrário das métricas baseadas em estatísticas. Assim, as características da *workload*, tais como ser dominada por programas muito longos ou ainda possuir valores extremos que afetem a média aritmética dos *turn-around times* não

interferem no resultado desta métrica, uma vez que a comparação é feita programa a programa.

Este mecanismo de avaliação permite que se conheça exatamente em quanto um algoritmo de escalonamento foi mais eficiente do que outro para uma *workload*, além de permitir identificar para quais programas. A partir destas informações é possível isolar o conjunto de programas onde cada algoritmo foi melhor, a fim de identificar e estudar suas características, como veremos adiante no estudo de caso.

Como desvantagem, podemos citar que para comparar mais de dois escalonadores é necessário fazê-lo dois a dois, obtendo sempre a relação de um *turnaround time* para com outro.

4.2. *Easy* × *Conservative Backfilling*

Nosso estudo de caso investiga o controvertido problema *Easy backfilling* × *Conservative backfilling* [25] [42]. Os artigos [25] e [42] analisam a performance de computadores IBM SP2 utilizando ambas as formas de *Backfilling*: *Easy* e *Conservative*. No artigo [25] conclui-se que a performance dos dois algoritmos é similar, sendo que *Conservative* possui a vantagem da previsibilidade. Em [42], chega-se à conclusão que, de forma geral, as *workloads* provenientes dos computadores SP2 favorecem o *Easy backfilling*. Porém em algumas métricas o resultado parece se inverter. Fica no ar, portanto, a pergunta: Qual métrica vale de fato?

Estes resultados controvertidos e inconclusivos hoje conhecidos nos motivaram a utilizar esta nova maneira de comparar desempenho, Performance Relativa, a fim de avaliarmos o mesmo problema.

Realizamos a simulação dos métodos de escalonamento *Conservative backfilling* e *Easy backfilling* a fim de efetuar o processo de avaliação de performance destes algoritmos através de algumas métricas conhecidas e também da Performance Relativa. Foram usados três *logs* de *workloads* provenientes de diferentes centros de supercomputação disponíveis na *web* [31] [44] e ainda uma quarta *workload* sintética obtida através de um modelo de carga [10].

Simulação

O nosso simulador recebe como entrada uma *workload* contendo para cada programa a quantidade de processadores (*n*), o *request time* (*tr*), o momento da submissão e o tempo de execução (*te*). Também são fornecidos ao simulador o método de escalonamento e a descrição do supercomputador, incluindo o número de processadores disponíveis.

O processamento destes programas é simulado através dos métodos de escalonamento *Easy backfilling* e *Conservative backfilling* obtendo-se os *turn-around times* como resultado. Estes *turn-around times* são então comparados através das métricas:

- *Turn-around time* médio (*tt* médio);
- *Slowdown* médio;
- *Bounded slowdown* médio com $\tau = 10$;
- *Bounded slowdown* médio com $\tau = 100$;
- Média geométrica e
- Performance relativa.

Workloads

Com o objetivo de tornar a simulação o mais realista possível, foram utilizados *traces* de supercomputadores de diferentes fontes disponíveis na *web* [31] [44], além de um modelo sintético [10].

nome	máquina	processadores	programas	período
CTC	*Cornell Theory Center SP2*	430	79.296	Jul 1996 Mai 1997
KTH	*Swedish Royal Institute of Technology SP2*	100	28.474	Set 1996 Ago 1997
SDSC	*San Diego Supercomputer Center SP2*	128	16.376	Jan 1999 Mai 1999
SYNT	Modelo Sintético	800	50.000	-

Tabela 4 - Descrição das *workloads*

Nem todos os programas que são submetidos aos supercomputadores completam a sua execução. Alguns são cancelados pelo usuário. Como estamos anali-

sando as métricas baseadas nos *turn-around times* dos programas, retiramos aqueles cancelados das *workloads* CTC, SDSC e do modelo sintético a fim de considerarmos só os programas que completaram sua execução ou foram interrompidos pelo escalonador por exceder o tempo requisitado (*tr*). A *workload* KTH não faz discriminação entre os programas cancelados e os completados.

4.3. Métricas Tradicionais

A Tabela 5 mostra os resultados de performance obtidos para cada método de escalonamento avaliado segundo as diversas métricas. Os valores desta tabelas estão expressos em segundos; a coluna EASY mostra o valor para o método *Easy backfilling* e a coluna CONS, *Conservative backfilling*.

	Média Geométrica		***Turn Around Time Médio***		***Slowdown Médio***		***Bounded Slowdown Médio $\tau = 10$***		***Bounded Slowdown Médio $\tau = 100$***	
Dados	EASY	CONS	EASY	CONS	EASY	CONS	EASY	CONS	EASY	CONS
CTC	1846	2323	11755	13409	5.32	13.97	5.32	13.97	4.36	8.60
KTH	1628	2133	14893	16202	158.11	209.35	79.44	92.26	20.55	20.82
SDSC	1474	1738	20158	20008	59.50	61.46	59.50	61.46	43.60	38.28
SYNT	1483	1561	10321	10695	67.99	42.52	25.36	16.47	7.65	5.64

Tabela 5 – *Easy* × *Conservative backfilling* com métricas tradicionais

Através da métrica *turn-around time* médio, o método *Conservative backfilling* foi ligeiramente melhor para a *workload* SDSC enquanto *Easy backfilling* foi melhor para as outras *workloads*. Usando a métrica média geométrica, *Easy* obteve melhor performance em todas as *workloads*. Já através da métrica *slowdown* médio, o método *Conservative* foi melhor para a *workload* sintética, enquanto *Easy* foi melhor para as outras *workloads*.

Inicialmente o *bounded slowdown* foi calculado com $\tau = 10$; como tanto para CTC como para SDSC não havia programas com *turn-around time* menores que 10, foi calculado novamente o *bounded slowdown* com $\tau = 100$. Através do *bounded slowdown médio* com $\tau = 10$, *Conservative* foi melhor para a *workload* sintética,

enquanto *Easy* foi melhor para as outras *workloads*. Já através do *bounded slowdown médio* com $\tau = 100$, *Conservative* foi melhor para a *workload* sintética e para SDSC, enquanto *Easy* foi melhor para CTC e ligeiramente melhor para KTH.

Percebemos que para estas *workloads* as avaliações de performance **apresentaram resultados conflitantes dependendo da métrica utilizada**. Estes conflitos reforçam outros resultados semelhantes na literatura [20] [25] [42].

4.4. Performance Relativa

Com o objetivo de contribuir para esclarecer melhor estes resultados contraditórios, utilizamos então a performance relativa. O cálculo foi feito segundo a razão $tt_{conservative}$ / tt_{easy} obtida a partir dos *turn-around times* dos programas submetidos aos métodos de escalonamento *Conservative backfilling* ($tt_{conservative}$) e *Easy backfilling* (tt_{easy}). Os gráficos das figuras 14 a 17 foram traçados a partir da distribuição acumulada de freqüência destes valores.

A porção da curva à esquerda de 1 representa onde o método *Conservative backfilling* foi melhor; a porção da curva à direita de 1 indica onde o método *Easy backfilling* foi melhor. Os valores 1 indicam onde os dois métodos obtiveram o mesmo *turn-around time.*

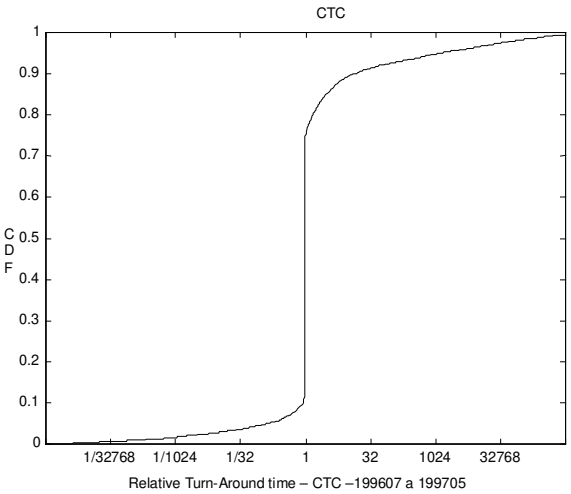

Figura 11 - Performance relativa CTC

Na Figura 11 vemos que para a *workload* CTC o método *Conservative* foi melhor em aproximadamente 12% dos *turn-around times*, enquanto *Easy* foi melhor em aproximadamente 25%. Outra análise importante que pode ser feita a par-

tir do gráfico é o quanto um algoritmo foi melhor em relação ao outro. Por exemplo, na *workload* CTC da Figura 11 cerca de 1% dos programas chegaram a concluir em tempo 32.000 vezes menor quando usando o escalonador *Conservative*. Por outro lado, usando o escalonador *Easy* alguns programas chegaram a executar em tempo 1.000.000 de vezes menor do que o escalonador usando *Conservative backfilling*.

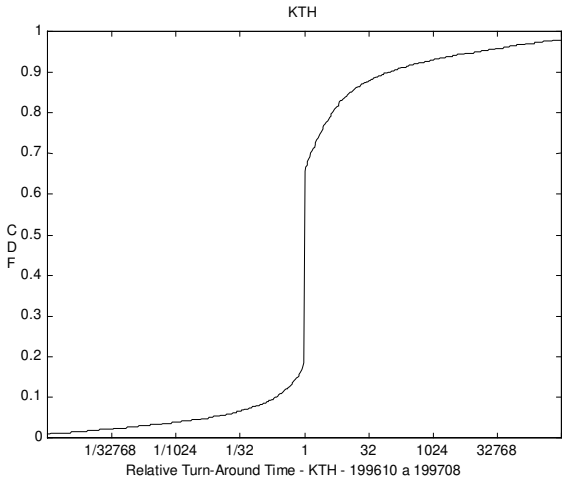

Figura 12 - Performance relativa KTH

Na *workload* KTH da Figura 12 percebemos que o método *Conservative* foi melhor em aproximadamente 18% dos *turn-around times*, enquanto *Easy* foi melhor em aproximadamente 35%. Em aproximadamente 4% dos programas, *Easy* foi mais de 32.000 vezes melhor, enquanto *Conservative* foi 32.000 vezes melhor em cerca de 2% dos casos.

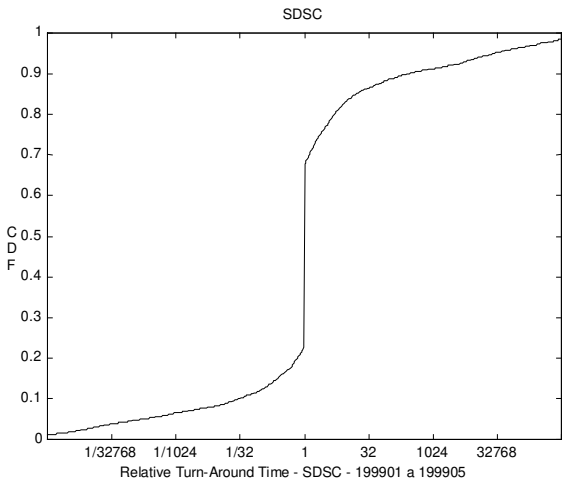

Figura 13 - Performance relativa SDSC

Na *workload* SDSC da Figura 13 temos que o método *Conservative* foi melhor em aproximadamente 23% dos *turn-around times*, enquanto *Easy* foi melhor em aproximadamente 33%.

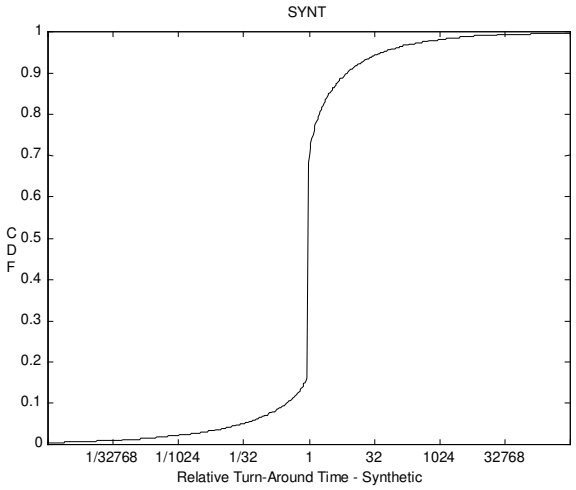

Figura 14 - Performance relativa SYNT

Na Figura 14 vemos que para a *workload* sintética o método *Conservative* foi melhor em aproximadamente 18% dos *turn-around times e Easy* foi melhor em 35%. O gráfico possibilita visualizar a relação entre os dois métodos: aqui, embora *Easy* tenha sido melhor em cerca de 35% dos casos, a vantagem não foi tão grande. A maioria dos programas foi apenas 32 vezes melhor, sendo que apenas cerca de 1% foi mais de 1.000 vezes melhor.

Analisando o conjunto das quatro *workloads*, percebemos um outro resultado surpreendente. A grande quantidade de programas com o mesmo *turn-around time* nos dois métodos, variando de 44% em SDSC até 63% em CTC.

4.5. Analisando as Características dos Programas

A segunda etapa desta análise de performance compreendeu o estudo das características dos programas, a fim de identificar quais características eram favorecidas por um determinado método de escalonamento. As características estudadas foram as seguintes:

- Número de processadores (n);
- Tempo requisitado (tr);
- Área requisitada ($tr \times n$);
- Tempo de execução (te) e
- Área de execução ($te \times n$).

Em cada *workload,* os programas foram ordenados por cada característica e divididos em três subconjuntos de igual tamanho: pequenos, médios e grandes. Para cada subconjunto traçamos então a Performance Relativa baseada nos *turn-around times* destes programas.

Número de processadores (n)

Para este estudo, classificamos os programas por número de processadores e traçamos a performance relativa para os três subconjuntos: pequenos, médios e grandes. O resultado pode ser visto na Figura 15 e nas tabelas 6 a 9: *Easy* mostrou melhor performance em todos os subconjuntos das *workloads* estudadas.

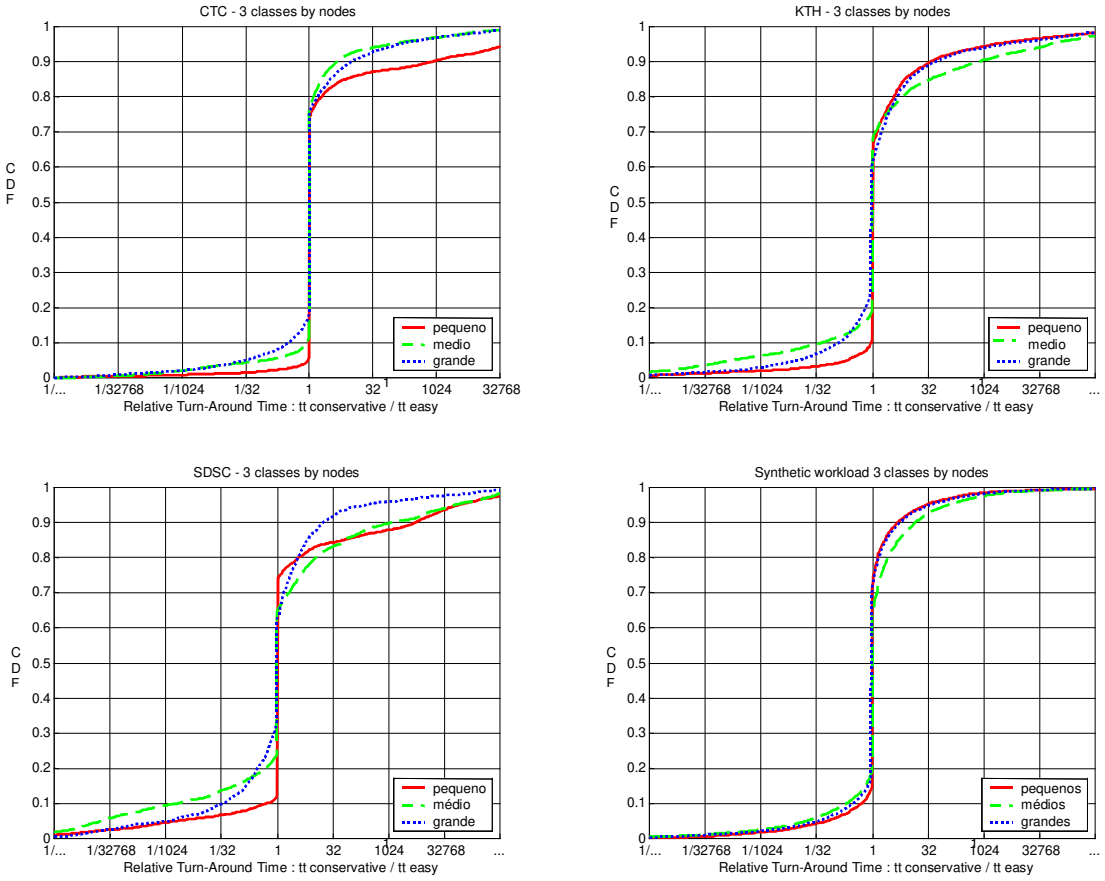

Figura 15 - Performance relativa das quatro *workloads* por *n*

Dados	EASY	CONS	IGUAL
Pequenos	27.079	5.64	67.27
Médios	24.60	11.77	63.61
Grandes	25.31	18.63	56.04

Tabela 6 - Sumário por *n* CTC

Dados	EASY	CONS	IGUAL
Pequenos	34.34	11.05	54.59
Médios	32.01	18.93	49.04
Grandes	39.29	25.77	34.92

Tabela 7 - Sumário por *n* KTH

Dados	EASY	CONS	IGUAL
Pequenos	26.27	12.32	61.40
Médios	35.51	24.32	40.16
Grandes	37.71	28.09	34.19

Tabela 8 - Sumário por *n* SDSC

Dados	EASY	CONS	IGUAL
Pequenos	28.62	15.94	55.43
Médios	37.32	19.70	42.96
Grandes	29.57	18.65	51.77

Tabela 9 -Sumário por *n* SYNT

Além disso, podemos perceber que, para *n* pequeno, o método *Easy* apresenta performance bem melhor. Tal resultado faz sentido. Afinal, é fácil perceber que um programa que solicita poucos *nodes* é mais fáceis de fazer *backfilling* que um programa que solicita muitos *nodes*. Entretanto, isto é verdade para ambos os métodos de escalonamento (*Easy* e *Conservative*). Então, porque *Easy* obtém uma performance superior a *Conservative* para programas que solicitam poucos *nodes*? Acreditamos que, quando se usa *Conservative*, nem sempre podemos aproveitar a oportunidade de fazer *backfilling* em programas pequenos. Em *Conservative*, pode haver um programa grande antes do pequeno, que embora não seja o primeiro da fila, impede o avanço do pequeno. Lembre-se que *Easy* permite adiantar um programa da fila desde que não atrase o início estimado do primeiro programa. Quando é possível realizar *backfilling*, *Easy* pode pegar o primeiro programa que usa os processadores livres e não atrasa o primeiro da fila e encaixar no *slot* vago. *Easy* pode "empurrar" os outros programas para trás, se o *tr* do programa que está sendo encaixado assim o exigir. *Conservative* não pode. Dessa forma, *Easy* consegue encaixar mais facilmente os programas com *n* pequeno. É interessante notar também que o fato de *Easy* beneficiar os programas com *n* pequeno não afeta negativamente os programas que têm *n* médio e grande. A performance de *Easy* é melhor também para os programas médios e grandes com relação a *n*. Suspeitamos que isto ocorre porque os tempos requisitados (*tr*)

são estimativas ruins dos tempos efetivamente usados na execução dos programas (*te*) [10]. Assim sendo, nas muitas vezes em que *Easy* "empurra" um programa para trás, este programa acaba não sendo prejudicado pois aquele programa que o "empurrou" termina muito antes do tempo requisitado.

Tempo requisitado (*tr*)

Assim como fizemos com *n*, a análise da performance em relação ao tempo requisitado (*tr*) foi feita avaliando a performance relativa dos três subconjuntos de programas divididos segundo esta característica (*tr*).

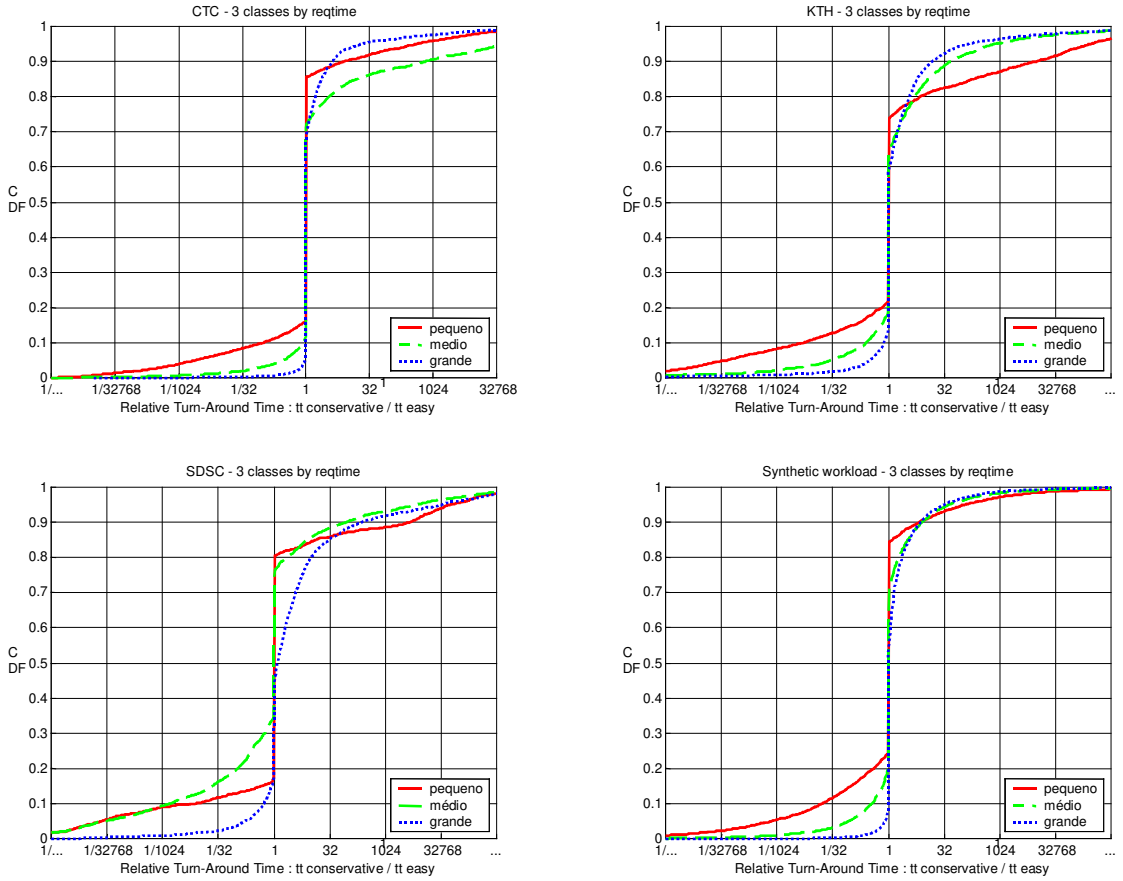

Figura 16 - Performance relativa das quatro *workloads* por *tr*

Dados	EASY	CONS	IGUAL
Pequenos	14.60	16.41	68.97
Médios	28.72	12.06	59.20
Grandes	33.67	7.57	58.75

Tabela 10 - Sumário por *tr* CTC

Dados	EASY	CONS	IGUAL
Pequenos	26.60	21.83	51.56
Médios	36.74	19.24	44.01
Grandes	42.31	14.68	42.99

Tabela 11 - Sumário por *tr* KTH

Dados	EASY	CONS	IGUAL
Pequenos	19.57	17.07	63.34
Médios	24.18	35.03	40.78
Grandes	55.73	18.73	25.52

Tabela 12 - Sumário por *tr* SDSC

Dados	EASY	CONS	IGUAL
Pequenos	15.88	24.24	59.87
Médios	29.85	20.11	50.02
Grandes	49.78	9.93	40.27

Tabela 13 - Sumário por *tr* SYNT

Através da Figura 16 e das tabelas 10 a 13, podemos perceber uma diferença entre a performance relativa dos programas com *tr* grande e a performance relativa dos programas com *tr* pequeno e médio. Enquanto para os programas com *tr* pequeno *Easy* e *Conservative* são mais semelhantes, *Easy* foi muito melhor para os programas grandes do que *Conservative*. Exceto por SDSC, também podemos notar que a performance de *Conservative* diminui à medida que os programas crescem. *Conservative* é pior para os grandes. Ou seja, a diferença de performance entre *Easy* e *Conservative* parece aumentar à medida que *tr* aumenta.

Este fenômeno pode ser explicado porque ao realizar *backfilling* quando *tr* é grande, *Easy* pode "abrir espaço" para encaixar um programa, mas *Conservative* não pode. *Conservative* só consegue aproveitar os espaços originalmente deixados pelo programa que terminou antes do tempo requisitado. Veja o exemplo da Figura 9, onde *Easy* só consegue adiantar o programa E porque pode "abrir espaço". En-

tão efetivamente *Conservative* tem um mecanismo de ação mais difícil de encaixar os programas com *tr* grande.

De maneira similar ao observado na análise de performance por *n*, o benefício que *Easy* traz para os programas com *tr* grande não se traduz em desvantagem para os programas com *tr* pequeno. Novamente, suspeitamos que isto ocorre porque *tr* é tipicamente uma má estimativa de *te*.

Observamos também uma diferença entre a curva de performance relativa dos programas pequenos e a curva de performance relativa dos programas médios e grandes: a curva é mais abrupta para os programas pequenos. Ou seja, a atuação do escalonador tem maior impacto sobre programas pequenos. Para os programas médios e grandes a curva é mais suave, indicando que o escalonador tem menos influência sobre os programas maiores. Acreditamos que este fenômeno se deve a dois fatores. Primeiro, programas menores são mais fáceis de fazer *backfilling* e portanto tem mais chance de se beneficiar do escalonador. Segundo, redução do tempo de espera (*tw*) tem mais impacto em programas menores. Por exemplo, uma otimização de 1 hora em um programa que tem *turn-around time* de 20 horas é muito menos impactante do que uma otimização de 1 hora em um programa que tem *turn-around time* de 2 horas.

É interessante notar que a performance relativa foi fundamental para que pudéssemos identificar o fenômeno acima descrito. Sumarizações da distribuição (como, por exemplo, as das Tabelas 10 a 13) tendem a esconder este fenômeno.

Área requisitada ($tr \times n$)

Assim como as características anteriores, o estudo do impacto da área requisitada sobre o escalonador foi feito classificando os programas por $tr \times n$ e traçando a performance relativa dos programas pequenos, médios e grandes. *Easy* obteve melhor performance para a maioria dos subconjuntos estudados.

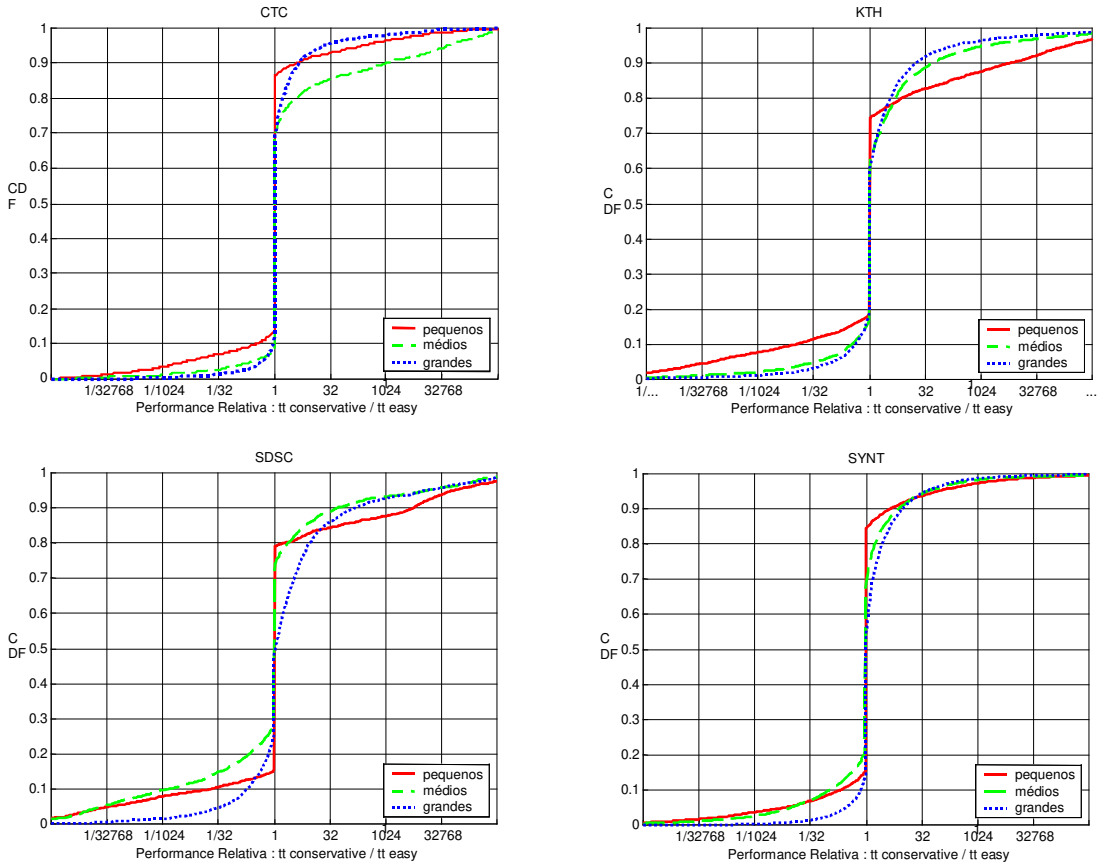

Figura 17 - Performance relativa das quatro *workloads* por *tr* × *n*

Dados	EASY	CONS	IGUAL
Pequenos	13.63	13.51	72.84
Médios	30.84	9.59	59.56
Grandes	32.52	12.94	54.53

Tabela 14 – Sumário por área requisitada CTC

Dados	EASY	CONS	IGUAL
Pequenos	25.92	18.51	55.55
Médios	38.39	17.19	44.41
Grandes	41.34	20.05	38.60

Tabela 15 – Sumário por área requisitada KTH

Dados	EASY	CONS	IGUAL
Pequenos	20.95	15.27	63.77
Médios	26.53	29.19	44.27
Grandes	52.00	26.38	21.61

Tabela 16 – Sumário por área requisitada SDSC

Dados	EASY	CONS	IGUAL
Pequenos	15.68	15.48	68.82
Médios	31.48	22.97	45.53
Grandes	48.34	15.84	35.81

Tabela 17 – Sumário por área requisitada SYNT

A área requisitada corresponde a $tr \times n$. Esta é a informação que o escalonador usa para alocar os programas. Observando o resultado da Figura 17 e das tabelas 14 a 17, percebemos um comportamento semelhante para todas as *workloads* utilizadas: os programas com grande área requisitada foram beneficiados pelo método *Easy*. Esta constatação justifica-se devido à maior possibilidade dos programas grandes serem adiantados na fila através do algoritmo *Easy* do que pelo *Conservative*. Temos quatro situações para $tr \times n$: (i) Se o programa tiver um n pequeno e um tr pequeno, teoricamente está na situação mais fácil de ser encaixado (realizar *backfilling*) pelos dois métodos; (ii) Se o programa tiver um n pequeno e um tr grande, os dois métodos conseguem encaixá-lo mais facilmente com relação a dimensão n, mas *Easy* leva vantagem porque tem maior possibilidade de "abrir espaço" para o tr grande, isto é, pode atrasar os programas da fila (exceto o primeiro), enquanto *Conservative* não pode abrir espaço; (iii) Se o programa tiver um n grande e um tr pequeno, é difícil para os dois métodos encaixá-lo, pois vai depender primeiramente de haver *nodes* disponíveis; (iv) Se o programa tiver um n grande e um tr grande, embora seja difícil de adiantá-lo, pois primeiro precisa haver *nodes* disponíveis, *Easy* terá mais facilidade de encaixá-lo, pois pode "abrir espaço" para a dimensão tr, atrasando os outros programas da fila (exceto o primeiro). *Conservative* não pode "abrir espaço", mas apenas aproveitar os espaços livres deixados por um programa que terminou antes do tempo requisitado. Mais ainda, de-

vido ao fato de que *tr* é comumente uma má estimativa de *te*, "abrir espaço" parece não ter grande impacto negativo no programa que foi "atrasado". Em resumo, a vantagem está com *Easy*.

Note também que os resultados por *tr* x *n* se parecem mais com os resultados por *tr*, que com os resultados por *n*. Isto reforça nossa análise de que o fato de *Easy* ter mais liberdade para agir com a dimensão *tr* dos programas é fundamental para a vantagem que *Easy* tem em relação a *Conservative*. Assim como observado nos resultados por *tr*, a curva de performance relativa é mais abrupta para os programas pequenos, indicando que o escalonador tem maior impacto sobre eles. Para os programas médios e grandes a curva é mais suave, mostrando que o impacto da ação do escalonador é menor.

Tempo de execução (*te*)

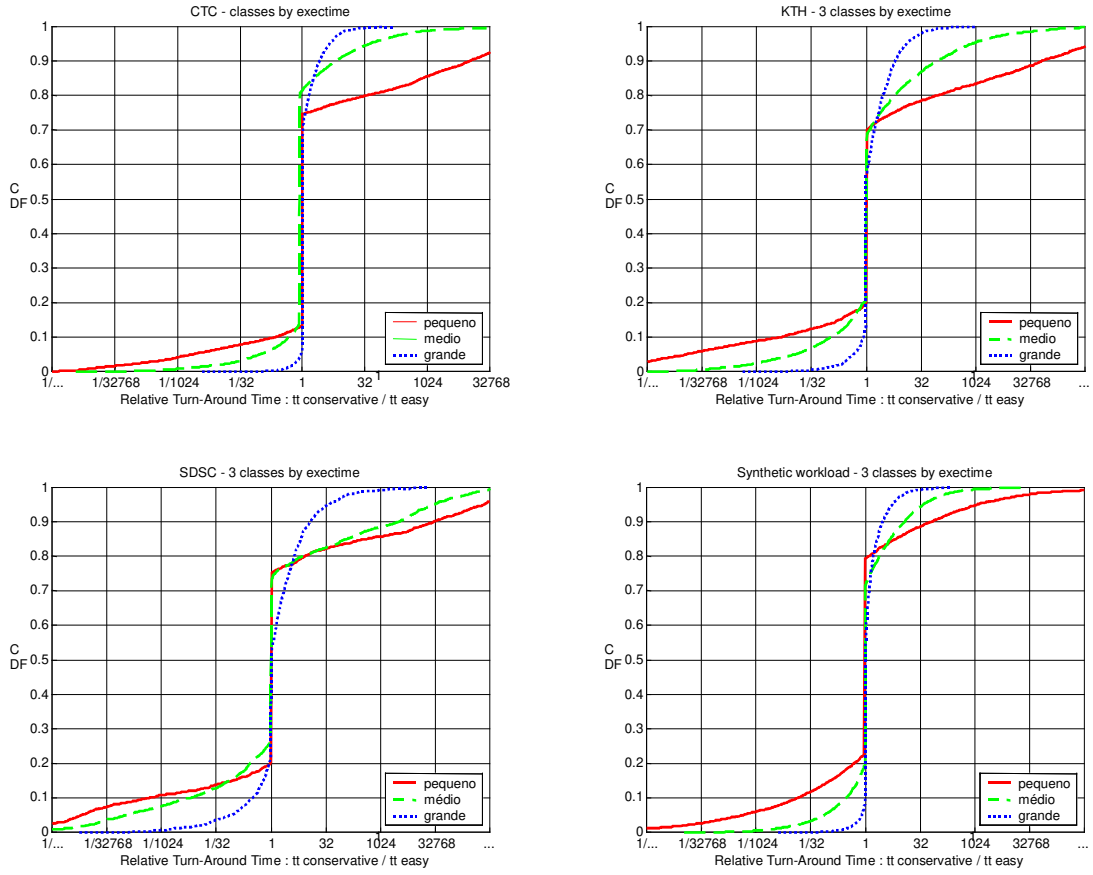

Figura 18 - Performance relativa das quatro *workloads* por *te*

Tal qual as demais características, o estudo do impacto do tempo de execução sobre o escalonador foi feito classificando os programas por *te* e traçando a per-

formance relativa dos programas pequenos, médios e grandes. O método de escalonamento *Easy backfilling* foi melhor para a grande maioria dos subconjuntos.

Dados	EASY	CONS	IGUAL
Pequenos	25.35	13.39	61.24
Médios	19.57	14.02	66.40
Grandes	32.07	8.63	59.28

Tabela 18 - Sumário por *te* CTC

Dados	EASY	CONS	IGUAL
Pequenos	30.43	19.66	49.89
Médios	31.93	21.48	46.58
Grandes	43.28	14.62	42.08

Tabela 19 - Sumário por *te* KTH

Dados	EASY	CONS	IGUAL
Pequenos	24.82	20.45	54.72
Médios	26.57	26.55	46.86
Grandes	48.08	23.84	28.06

Tabela 20 - Sumário por *te* SDSC

Dados	EASY	CONS	IGUAL
Pequenos	20.58	22.65	56.75
Médios	28.63	20.05	51.31
Grandes	46.29	11.59	42.10

Tabela 21 - Sumário por *te* SYNT

Observando o resultado na Figura 18 e nas tabelas 18 a 21, vemos que *Easy* foi muito melhor para os programas grandes segundo *te*. O tempo de execução é um fator correlacionado com o tempo requisitado, pois $te \leq tr$. Se *te* é grande, significa que *tr* é grande também (embora o contrário possa não ser verdadeiro). Mas, como vimos acima, *Easy* favorece programas com grande *tr*. Assim sendo,

Easy favorece os programas com grande *te*, uma vez que estes também têm grande *tr*.

Observa-se ainda a curva mais brusca para os programas pequenos e mais suave para os programas grandes, indicando que a mudança de escalonador é mais impactante para os programas pequenos. Quando primeiro observamos que a curva para programas pequenos é mais brusca (nos resultado por n), conjeturamos que isso tinha duas razões. Primeiro, programas menores são mais fáceis de fazer *backfilling* e portanto tem mais chance de se beneficiar do escalonador. Segundo, a redução do tempo de espera (*tw*) tem mais impacto em programas menores. Entretanto, programas com pequeno *te* não necessariamente são mais fáceis de escalonar. Programas com pequeno *te* podem ter grande *tr* e serem mais difíceis de escalonar. Assim sendo, a manifestação do fenômeno de curvas bruscas também para programas com pequeno *te* indica que a segunda razão levantada para explicar o fenômeno é realmente crucial para explicar os resultados observados.

Área de execução ($te \times n$)

O estudo da área de execução (te x n) também foi feito dividindo-se os programas em três subconjuntos: pequenos, médios e grandes.

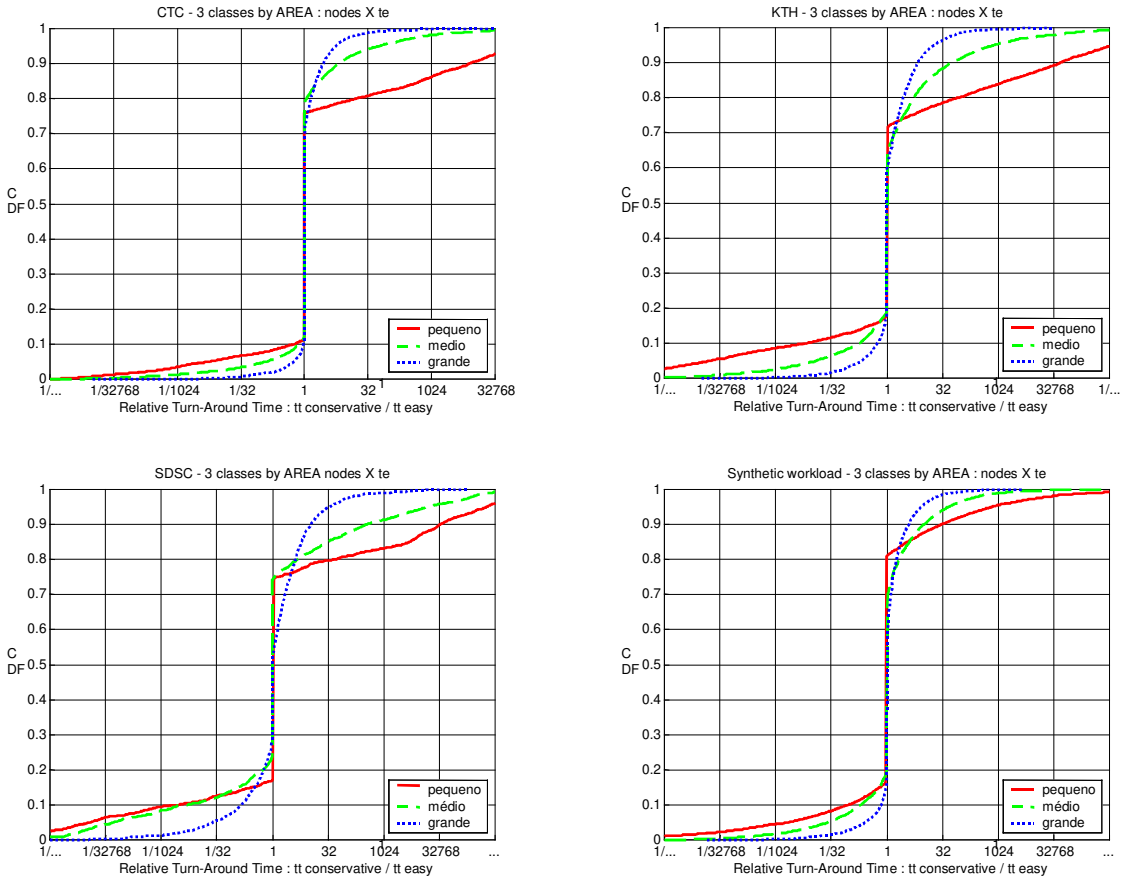

Figura 19 - Performance relativa das quatro *workloads* por *te* × *n*

Dados	EASY	CONS	IGUAL
Pequenos	24.16	11.37	64.45
Médios	21.72	12.47	65.80
Grandes	31.11	12.20	56.68

Tabela 22 - Sumário por *te* × *n* KTH

Dados	EASY	CONS	IGUAL
Pequenos	28.73	17.47	53.78
Médios	35.96	19.48	44.55
Grandes	40.97	18.80	40.22

Tabela 23 - Sumário por *te* × *n* KTH

Dados	EASY	CONS	IGUAL
Pequenos	25.48	17.41	57.10
Médios	25.81	23.72	50.45
Grandes	48.18	29.70	22.10

Tabela 24 - Sumário por *te* × *n* SDSC

Dados	EASY	CONS	IGUAL
Pequenos	18.91	16.40	64.67
Médios	31.33	20.55	48.11
Grandes	45.26	17.34	37.39

Tabela 25 - Sumário por *te* × *n* SYNT

A partir dos resultados da Figura 19 e das tabelas 22 a 25, verificamos que *Easy* é melhor para todos os subconjuntos estudados. A área de execução está correlacionada com a área requisitada, uma vez que $te \times n$ é sempre menor ou igual a $tr \times n$. Além disso, sempre que $te \times n$ é grande, $tr \times n$ também o é, embora o contrário nem sempre aconteça. Na verdade a relação entre $te \times n$ e $tr \times n$ é a mesma relação observada entre te e tr. Assim sendo, da mesma maneira que os resultados por te acompanham os resultados por tr (como argumentamos acima), é natural esperar que os resultados por $te \times n$ também acompanhem $tr \times n$.

4.6. Conclusão

De modo geral, o método de escalonamento *Easy backfilling* proporcionou uma melhor performance para as *workloads* analisadas do que o método *Conservative backfilling*.

Observamos que o maior impacto individual foi do tr. Enquanto para os programas com tr pequeno *Easy* e *Conservative* são mais semelhantes, *Easy* foi muito melhor para os programas grandes com relação à tr do que *Conservative*. Ainda mais, a diferença de performance entre *Easy* e *Conservative* parece aumentar à medida que tr aumenta. Este fenômeno pode ser explicado porque ao realizar *backfilling* quando tr é grande, *Easy* pode "abrir espaço" para encaixar um programa, mas *Conservative* não pode. *Conservative* só consegue aproveitar os espa-

ços originalmente deixados pelo programa que terminou antes do tempo requisitado. Esta possibilidade de atrasar os programas parece ser muito importante, trazendo um diferencial positivo para o método de escalonamento *Easy backfilling*. É interessante notar que atrasar os programas da fila não parece gerar problemas para eles. Acreditamos que o os programas atrasados não sofrem devido ao fato que tr é uma má estimativa de te. Há estudos na literatura que mostram que, de modo geral, a estimativa de tr não é muito precisa (o que dá oportunidade para o escalonador agir) [10]. Portanto, o benefício que *Easy* traz para os programas com tr grande não se traduz em desvantagem para os programas com tr pequeno.

Percebemos também que, para n pequeno, o método *Easy* apresenta performance bem melhor. Como *Easy* permite adiantar um programa da fila desde que não atrase o início estimado do primeiro da fila, quando é possível realizar *backfilling Easy* pode pegar o primeiro programa que usa os processadores livres e não atrasa o primeiro da fila e encaixar no *slot* vago. Isto é, *Easy* pode "empurrar" os outros programas para trás, se o tr do programa que está sendo encaixado assim o exigir. Já em *Conservative*, pode haver um programa grande antes do pequeno, que embora não seja o primeiro da fila, impede o avanço do pequeno. *Conservative* não pode "empurrar" nenhum programa para trás. Dessa forma, *Easy* consegue encaixar mais facilmente os programas com n pequeno e tr grande. É interessante notar também que o fato de *Easy* beneficiar os programas com n pequeno não afeta negativamente os programas que têm n médio e grande. A performance de *Easy* é melhor também para os programas médios e grandes com relação a n. Novamente suspeitamos que isto ocorre porque os tempos requisitados (tr) são estimativas ruins dos tempos efetivamente usados na execução dos programas (te) [10]. Assim sendo, nas muitas vezes em que *Easy* "empurra" um programa para trás, este programa acaba não sendo prejudicado pois aquele programa que o "empurrou" termina muito antes do tempo requisitado.

Através da observação das curvas de performance relativa, verificamos também uma diferença entre a curva de performance relativa dos programas pequenos segundo tr, $tr \times n$, te e $te \times n$ e a curva de performance relativa dos programas médios e grandes: a curva é mais abrupta para os programas pequenos. Ou seja, a atuação do escalonador tem maior impacto sobre os programas pequenos segundo estas características. Para os programas médios e grandes a curva é mais suave,

indicando que o escalonador tem menos influência sobre os programas maiores. Acreditamos que este fenômeno se deve a dois fatores. Primeiro, programas menores são mais fáceis de fazer *backfilling* e portanto tem mais chance de se beneficiar da atuação do escalonador. Porém, programas com pequeno *te* não são necessariamente mais fáceis de escalonar. Programas com pequeno *te* podem ter grande *tr* e serem mais difíceis de escalonar. Então investigamos uma segunda explicação para o fenômeno e verificamos que a redução do tempo de espera (*tw*) tem mais impacto em programas menores, uma vez que $tt = tw + te$. A manifestação do fenômeno de curvas bruscas também para programas com pequeno *te* indica que esta segunda razão é realmente crucial para explicar os resultados observados.

Acreditamos que estes resultados podem ser estendidos a outras *workloads* que utilizam os métodos de escalonamento *Easy backfilling* e *Conservative backfilling*. Esta afirmação se baseia no fato do estudo de caso contemplar um conjunto de *workloads* bastante representativo, pois além de incluir um modelo sintético, usa também três *workloads* reais.

Mas a principal conclusão é sobre o uso das métricas de performance. As métricas baseadas em estatísticas sumarizadoras podem levar a resultados enganosos. Vimos em 4.3 que os resultados da avaliação de performance são contraditórios, principalmente aqueles baseados em *slowdown* e *bounded-slowdown*. Mas até mesmo a média aritmética dos *turn-around times* apresentou um resultado divergente. A métrica sumarizadora média geométrica concordou com os resultados obtidos através de performance relativa. A performance relativa permitiu comparar os *turn-around times* de toda a distribuição, mostrando mais claramente como foi a diferença de performance entre os métodos. Além disso, permitiu-nos analisar os subconjuntos dos programas, a fim de investigar as relações entre o método de escalonamento e as características das *workloads*.

Contudo, não existe uma métrica perfeita. Ao realizar a avaliação de performance, é importante lançar mão de várias métricas que permitam conhecer o que está acontecendo entre a *workload* e o método de escalonamento. Caso haja divergências nos resultados, é interessante analisar o porquê do conflito. O uso de performance relativa não sumariza os resultados, e por isso não “esconde” o que está acontecendo com os dados. Por isso achamos recomendável aplicá-la na avaliação de performance.

Capítulo V

5. Escalonamento e Moldabilidade

Os supercomputadores paralelos são tipicamente *space-shared*, isto é, cada programa recebe uma partição (um subconjunto dos processadores) para executar [9]. O usuário do supercomputador informa a quantidade de processadores (n) e o tempo requisitado (tr) para a execução do programa. Este par (n,tr) caracteriza um *request* estático. Atualmente os escalonadores de supercomputadores recebem apenas *requests* estáticos (Easy [35], PBS [32], Maui [40], LSF [46], CRONO [43], entre outros).

Um programa moldável é aquele que pode ser executado em partições de diferentes tamanhos, mas uma vez iniciada a execução, o tamanho da partição não pode ser alterado [26]. A maioria dos programas paralelos são moldáveis: uma pesquisa realizada com usuários de supercomputadores constatou que a maioria dos programas paralelos pode ser executado com diferentes *requests* [14]. Veja na Figura 20 o resultado da pesquisa quando foi perguntado quantas partições diferentes os usuários já requisitaram para sua aplicação. Note que, quando o programa é moldável, o usuário precisa escolher qual *request* utilizar. A Figura 21 mostra os usuários escolhendo a opção de submissão dos programas moldáveis.

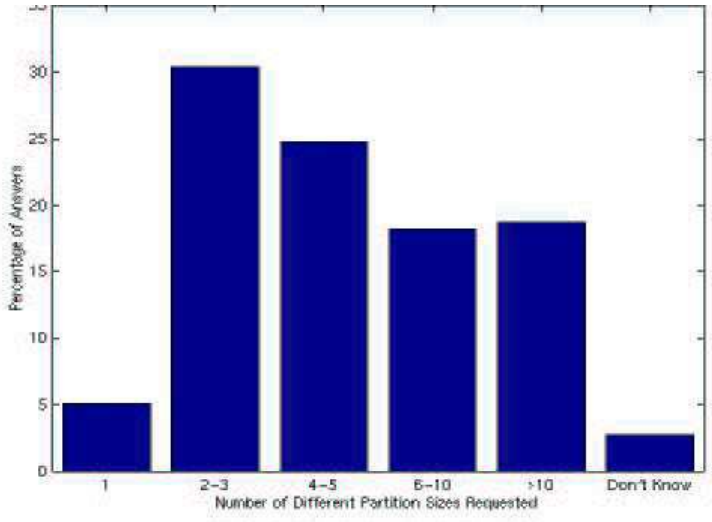

Figura 20 - Moldabilidade dos programas paralelos [14]

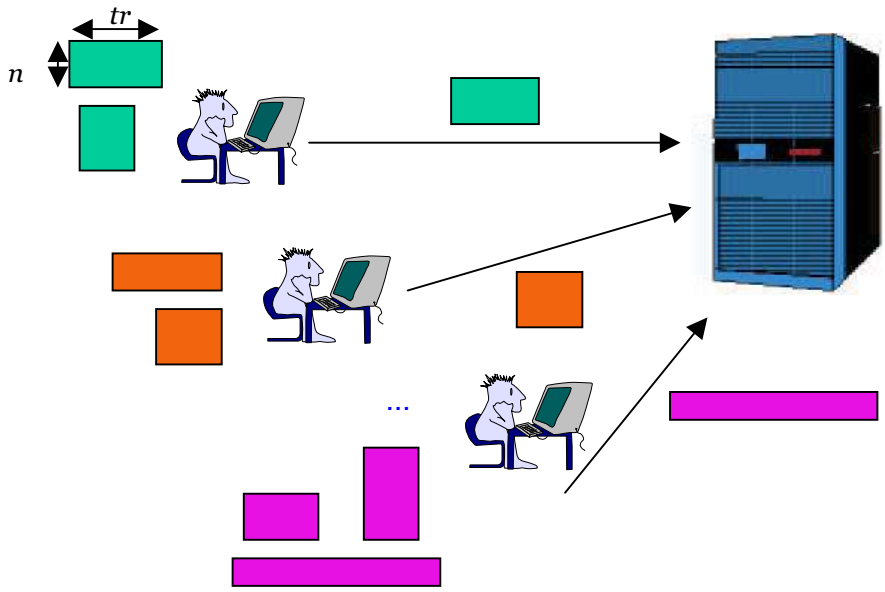

Figura 21 - Seleção dos *requests* pelos usuários [9]

A escolha de (*n*,*tr*) é importante porque afeta diretamente o *turn-around time* do programa, isto é, o tempo desde a sua submissão até o final da sua execução. Como já foi dito, o *turn-around time* é a soma do tempo de execução (*te)* mais o tempo de espera (*tw*). Em geral, quando se aumenta o número de processadores, o tempo de execução diminui. Por outro lado, o tempo de espera aumenta, pois como o programa precisa de mais processadores, freqüentemente terá que esperar mais na fila. Portanto, para que seu programa execute "o mais rápido possível", o usuário precisa estimar qual *request* (*n*,*tr*) trará uma melhor performance para o seu programa, escolhendo um número de processadores que não faça o programa ficar tanto tempo na fila esperando por recursos, mas que ao mesmo tempo proporcione um tempo de execução aceitável. Infelizmente, estimar o tempo de espera é algo bastante difícil, pois depende de *n*, *tr*, do escalonador e das condições de carga do supercomputador. Várias pesquisas que procuraram obter previsão do tempo de espera consideraram difícil conseguir uma boa previsão [19] [30] [49] [50].

5.1. SA

Para ajudar o usuário nesta difícil escolha de qual *request* selecionar para obter a melhor performance para a sua aplicação, foi desenvolvido SA (*Supercomputer AppLeS*) [9]. *AppLeS* são os escalonadores de aplicação desenvolvidos pelo

grupo de Francine Berman na UCSD [4] [5]. SA é um escalonador de aplicação que, a partir de um conjunto de possíveis *requests* fornecidos pelo usuário, seleciona um que freqüentemente reduz o *turn-around time* do programa, de acordo com experimentos realizados em [9]. A Figura 22 ilustra como SA trabalha.

SA avalia o conjunto de possíveis *requests* e faz uma estimativa do *turn-around time* de cada um, baseado na situação atual do supercomputador. A estimativa é feita assumindo que não há novos programas chegando ao sistema e que o tempo de execução dos programas é igual ao tempo requisitado, *tr*. Note que não há garantias de que SA selecionará o melhor *request*, mas isto acontece freqüentemente [9].

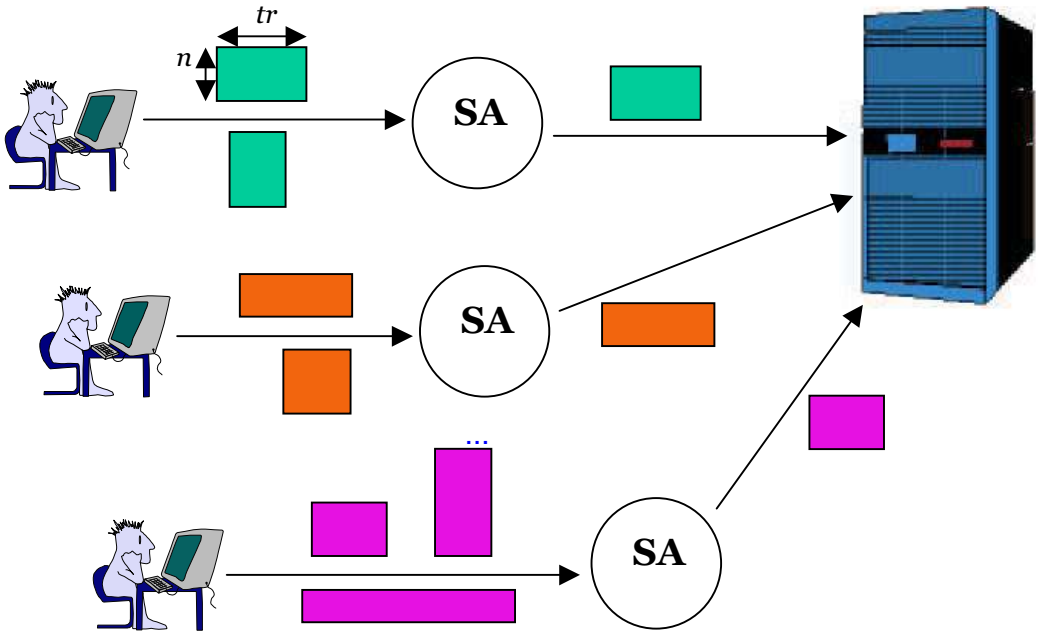

Figura 22 - SA selecionando o *request* [9]

Mais precisamente, a simulação da submissão do programa j com um determinado *request* $r^{[i]}$ é feita a partir do estado atual do supercomputador e é realizada através dos eventos de escalonamento, até que o programa j termine. Os eventos de escalonamento são submissões e términos. Apenas uma submissão é provida ao sistema: $r^{[i]}$, que submete o programa j. Os términos são providos para todos os programas no sistema, inclusive para o programa j. Os términos são calculados assumindo que cada programa executa pelo tempo requisitado *tr*. Ou seja, SA realiza a simulação do $r^{[i]}$ (i) assumindo que não há chegadas futuras de programas e (ii) fazendo $te = tr$ para todos os programas no sistema. Entretanto, na realidade, no-

vos programas chegam ao sistema depois do programa *j*. Além disso, a maioria dos programas executa por menos tempo do que requisitaram. Assim sendo, não há garantias de que SA irá selecionar o *request* que obterá o menor *turn-around time*. Entretanto, na prática os *requests* selecionados por SA melhoram significativamente o *turn-around time* em relação aos *requests* selecionados pelos usuários [9].

O uso intenso de SA afeta a natureza da carga que está sendo processada pelo supercomputador. Quando muitas instâncias de SA estão trabalhando em um supercomputador, as decisões tomadas por uma instância de SA afetam o estado do sistema, conseqüentemente influenciando nas decisões que as outras instâncias de SA estão tomando.

Embora seja verdade que esta competição torne mais difícil o trabalho de cada instância de SA em melhorar a performance do seu programa, existem outros comportamentos emergentes que surgem quando a carga varia de moderada a intensa. Em primeiro lugar, quando a carga aumenta, SA escolhe menores *requests*, aumentando a eficiência do sistema (uma vez que a maioria dos programas paralelos tem *speed-up* sublinear), o que reduz a carga oferecida e reduz longos *wait times*. Segundo, um melhor empacotamento dos programas e o fato de existirem menos programas no sistema tornam mais fácil para os programas que chegam na fila de espera serem escalonados, conseqüentemente reduzindo também os *wait times*. Em resumo, em condições de carga moderada a intensa, uma instância de SA beneficia-se do fato de existirem outras instâncias de SA no sistema [9] [11].

O bom comportamento emergente mostrado por SA levanta uma questão natural. O que acontece se a decisão da forma do programa moldável puder ser postergada para o momento do início da execução do programa? Este é um passo natural da investigação do impacto de SA sobre a performance. A idéia é que postergando o momento da ação de SA, seja possível tomar uma decisão melhor. Esta alteração envolve uma mudança no escalonador de recursos do supercomputador. Na versão original SA atua como um escalonador de aplicação. Nesta segunda versão, SA tem que ser embutido no escalonador do supercomputador. Para tanto, as várias alternativas de *request* para os programas moldáveis são informadas ao escalonador do supercomputador. No momento da submissão, SA é invocado, escolhendo o *request* a ser usado. Entretanto, sempre que há oportunidade de *backfilling*, SA é invocado para escolher novamente o *request* dentre todos os

requests possíveis para aquele programa. Então, baseado na situação atual do supercomputador, isto é, conhecendo os programas em execução, SA re-estima o melhor *request* para o programa que vai iniciar. A suposição é que esta decisão possa trazer uma performance ainda melhor do que SA atuando apenas no momento da submissão. A Figura 23 apresenta esquematicamente a modificação realizada.

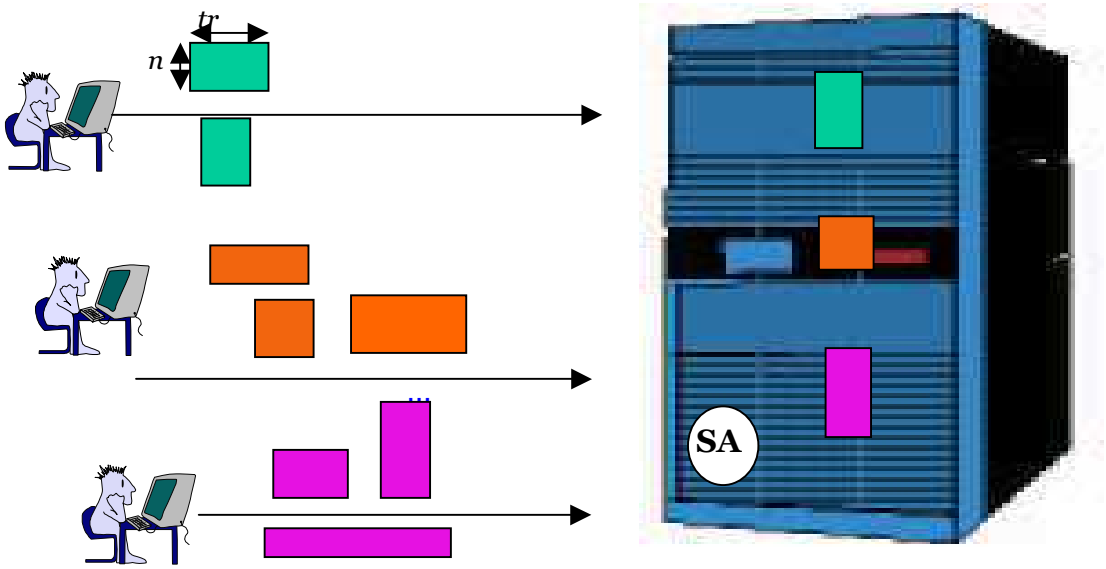

Figura 23 - SA decidindo no momento do início do programa

Para clareza da apresentação, vamos definir SA_s como sendo o SA original (o escalonador de aplicação que atua no momento da submissão) e SA_i para SA embutido no escalonador de recursos do supercomputador, que é invocado tanto na submissão do programa como sempre que há oportunidade de realizar *backfilling*.

Mais precisamente, em SA_i, os usuários informam diretamente ao escalonador do supercomputador o conjunto de *requests* $r = (r^{[1]},..., r^{[v]})$ que pode ser usado para submeter um programa j. No momento da submissão, SA_i é invocado e avalia o conjunto de *requests* r, escolhendo um deles para submeter o programa. Quando acontece uma oportunidade de realizar *backfilling* e o programa é adiantado na fila, podendo começar imediatamente, SA_i é invocado de novo. No momento do início do programa SA_i conhece a situação atual do supercomputador e usa esta informação para calcular o *request* que proporcionará o menor *turn-around time*. SA_i simula a execução de todos os *requests* em r, e então seleciona o *request* $r^{[s]}$ que consegue o menor *turn-around time* nas simulações.

Espera-se que esta modificação tenha uma conseqüência direta sobre a performance do programa que usa SA_i. Porém, provavelmente não só o programa que usa SA_i terá sua performance afetada. O sistema passará a comportar-se de forma diferente, pois com várias instâncias de SA_i decidindo no início dos programas, as características da *workload* são alteradas. A atuação de várias instâncias de SA_i gera um novo comportamento no sistema.

5.2. $SA_s \times SA_i$

Para que pudéssemos avaliar como SA_i se comporta na prática, foi necessário determinar um conjunto de programas moldáveis que correspondesse ao cenário de uma *workload* de supercomputador no mundo real. Utilizamos para tanto o modelo apresentado em [14]. Este modelo foi construído com base nas observações estatísticas de quatro logs de *workloads* e no resultado de uma pesquisa realizada com usuários de supercomputadores (essencial para obter informações sobre moldabilidade). As quatro *workloads* usadas como referência para o modelo de carga estão sumarizadas na Tabela 26. A descrição detalhada do modelo, assim como o questionário, podem ser encontrados em [9].

Nome	Máquina	Procs.	Programas	Período
ANL	*Argonne National Laboratory SP2*	120	7995	Out 1996 Dez 1996
CTC	*Cornell Theory Center SP2*	430	79279	Jul 1996 Mai 1997
KTH	*Swedish Royal Institute of Technology SP2*	100	28479	Set 1996 Ago 1997
SDSC	*San Diego Super-computer Center SP2*	128	16376	Jan 1999 Mai 1999

Tabela 26 - *Workloads* de referência para o modelo de carga [14]

O modelo de carga moldável é composto de duas partes independentes: o modelo de carga rígido [10] e o modelo de moldabilidade [14]. O modelo de carga rígido produz um conjunto de programas, cada um com um *request*. O modelo de moldabilidade gera *requests* alternativos para um dado programa moldável *j*, para o qual apenas um *request* é conhecido. A *workload* moldável é obtida usando primeiro o modelo de carga rígido para obter um conjunto de programas e então apli-

cando o modelo de moldabilidade para obter os *requests* adicionais para os programas.

SA_s foi avaliado usando *Conservative backfilling* como método de escalonamento por este ser um mecanismo representativo da prática de escalonamento em supercomputadores paralelos atualmente. Nosso simulador que implementa SA_i também foi avaliado com *Conservative backfilling*, a fim de compararmos os nossos resultados com os resultados obtidos anteriormente sobre a performance de SA_s e já disponíveis na literatura [9] [11].

Simulação

Realizamos um total de 23.692 experimentos para investigar a performance de SA nos cenários desejados. Começamos simulando um supercomputador com 500 processadores, e então simulamos também outros supercomputadores com 250 e 750 processadores, a fim de compararmos o comportamento de SA com diferentes cargas. A Tabela 27 mostra o número de experimentos por supercomputador. Para cada experimento, foi gerado um conjunto de 10.000 programas usando o modelo descrito.

processadores	experimentos
250	4348
500	14544
750	4800

Tabela 27 - Quantidade de experimentos por supercomputador

Cada experimento focalizou um determinado programa *target j* e o resultado foi o comportamento deste programa nos cenários (i) *j* com um *request* estático (ii) *j* com SA_s e (iii) *j* com SA_i. Os *requests* usados pelos outros programas que compõem a *workload* variaram em (i) *workload* usando o *request* estático (ii) *workload* usando SA_s e (iii) *workload* usando SA_i. Portanto, cada experimento envolveu nove simulações. A combinação de situações possíveis nos conduz aos seguintes cenários mostrados na Tabela 28.

CENÁRIOS	programa usa (*target*)	outros programas usam (*workload*)
target em uma *workload* estática	*request* estático	*request* estático
	SA_s	*request* estático
	SA_i	*request* estático
target em uma *workload* moldável com SA_s	*request* estático	SA_s
	SA_s	SA_s
	SA_i	SA_s
target em uma *workload* moldável com SA_i	*request* estático	SA_i
	SA_s	SA_i
	SA_i	SA_i

Tabela 28 - Cenários simulados com SA

Devido ao grande número de simulações e ao tempo demandado para processamento (para realizar um único experimento eram necessários aproximadamente 11 minutos em um computado Pentium III 1.2 Ghz e 128 Mb de RAM) utilizamos *MyGrid* a fim de acelerar a obtenção dos resultados. *MyGrid* é um ambiente de execução de aplicações paralelas *bag-of-tasks*, isto é, aplicações que podem ser executadas em qualquer ordem e não precisam ser executadas simultaneamente [13]. Esta é exatamente a característica da nossa aplicação.

5.3. Avaliação de Performance

A avaliação de performance que desejamos fazer sobre as alternativas de escalonamento de programas moldáveis procura responder inicialmente à seguinte questão. Se um usuário deseja obter a melhor performance para a sua aplicação, qual a melhor maneira de submeter o seu programa: usando um *request* estático (escolhido por ele), usando SA_s ou usando SA_i? Para responder esta questão estudamos o comportamento do *target* em três cenários: em uma *workload* estática, em uma *workload* onde todos os programas usam SA_s e em uma *workload* que usa SA_i. Isto é, deixamos a *workload* fixa e variamos o programa *target*. Em cada um destes cenários, o *target* variou entre usar o *request* estático, usar SA_s e usar SA_i. Constatamos que a para as três *workloads* performance obtida quando o programa *target* usa SA_i é melhor do que a performance obtida quando o programa *target* usa SA_s, que por sua vez é melhor do que a performance obtida quando o programa

target usa um *request* estático. Isto é, postergar a decisão tomada por SA para um momento mais próximo do início do programa traz uma melhor performance para o programa *target* (embora a melhoria que o uso de SA_i proporciona seja pequena em relação à performance obtida com o uso de SA_s). Isto responde a primeira pergunta: quando o usuário decide isoladamente, sem se preocupar com o que vai acontecer com o restante do sistema, é sempre melhor para o programa *target* usar SA_i.

A segunda questão da nossa avaliação de performance é: qual o efeito causado para o sistema quando todos os usuários tentam maximizar a performance de seus programas usando SA_i? Surpreendentemente, a performance obtida quando todos usam SA_i é pior do que a performance obtida quando todos usam SA_s, embora seja bem melhor do que quando todos usam o *request* estático. Isto é, a decisão tomada individualmente por cada usuário para obter a melhor performance possível acaba trazendo uma conseqüência de piorar o desempenho global. Acontece que SA_i tende a escolher o maior n, e com isso os programas que estão esperando pelos processadores livres tendem a esperar mais na fila, aumentando conseqüentemente o *turn-around time*. O benefício trazido pela melhoria de performance proporcionada pela oportunidade de escolher uma maior quantidade de recursos no momento do início da execução é neutralizada pelo aumento do *wait time*.

Concluímos que a melhor alternativa quando todos os programas estão usando moldabilidade é SA_s. SA_s obtém performance bem superior ao uso do *request* estático e além disso mostra um comportamento agregado benéfico em algumas situações, como visto em 5.1. Quando a carga aumenta, SA_s escolhe menores *requests*, aumentando a eficiência do sistema, uma vez que a maioria dos programas paralelos tem *speed-up* sublinear, o que reduz a carga oferecida e reduz longos *wait times*. Além disso, um melhor empacotamento dos programas e o fato de existirem menos programas no sistema tornam mais fácil para os programas que chegam na fila de espera serem escalonados, conseqüentemente reduzindo os *wait times*. Esta redução dos *wait times* melhora o *turn-around time* global.

A melhor alternativa quando um programa decide isoladamente obter melhor performance é SA_i. Porém, curiosamente não é a melhor alternativa para

quando todo o conjunto de programas procura obter melhor performance. Neste caso, o melhor é usar SA_s.

Programa *target* em *workload* estática

Chamamos a *workload* de estática quando os programas que compõem a *workload* usam apenas *requests* estáticos. Quando a *workload* é estática, o programa *target* pode: (i) usar um *request* estático (escolhido pelo usuário) (ii) usar SA_s ou (iii) usar SA_i. Vamos visualizar o comportamento do *target* nestas três situações. A performance relativa para 250, 500 e 750 processadores é bastante semelhante. Mostramos aqui a performance relativa traçada para o supercomputador com 500 processadores. A Figura 24 mostra a performance relativa obtida através da razão entre os *turn-around times* de:

- (ii) *target* usando SA_s em uma *workload* estática (sub-static) / (i) *target* estático em uma *workload* estática (static-static)
- (iii) *target* usando SA_i em uma *workload* estática (exec-static) / (i) *target* estático em uma *workload* estática (static-static)

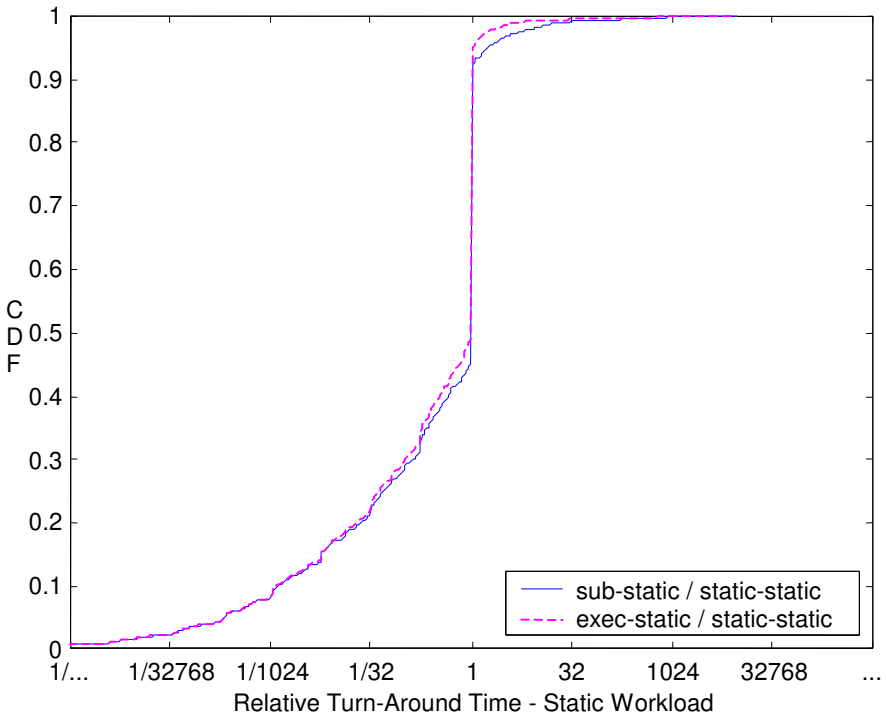

Figura 24 - Performance relativa - *workload* estática

Tomamos como base inicialmente *target* estático em uma *workload* estática. Quando o *target* usa SA_s obtém performance muito melhor do que quando o *target* usa um *request* estático. Da mesma maneira, quando o programa *target* usa SA_i também obtém um desempenho muito melhor do que quando o programa *target* usa um *request* estático. SA_i é melhor do que SA_s, mas não é por muito. Vejamos nas Tabelas 28 e 29 o sumário da performance relativa para a *workload* estática:

	%
target estático melhor	7.93
igual	46.39
target com SA_s melhor	45.66

Tabela 29 - Sumário de SA_S em *workload* estática

	%
target estático melhor	5.36
igual	44.94
target com SA_i melhor	49.68

Tabela 30 - Sumário de SA_i em *workload* estática

Percebemos que a performance é muito melhor quando se usa uma das formas de SA. Entretanto, é possível perceber que a diferença de performance entre usar SA_s ou SA_i não é tão grande. O fato de o *target* obter uma performance um pouco melhor quando usa SA_i pode ser explicado porque quando SA_i atua no momento da execução, ele conhece os recursos efetivamente disponíveis no sistema e pode escolher um *request* ainda melhor para a aplicação do que no momento da submissão. Todavia, não deixa de ser surpreendente que a melhoria não seja maior.

Vejamos na Tabela 31 a performance calculada através das diversas métricas de performance discutidas em 3.4.

		target		
	Métrica	Static	SA_s	SA_i
250 procs	*mean*	63326	46869	41926
	geomean	3978	1990	1856
	slowdown	162	106	80
	bounded 10	89	64	52
	bounded 100	39	29	22
	std deviation	215697	174066	165758
500 procs	*mean*	37977	32659	28196
	geomean	2999	1474	1396
	slowdown	139	283	221
	bounded 10	63	74	55
	bounded 100	26	29	20
	std deviation	113505	114619	99271
750 procs	*mean*	31265	27298	23344
	geomean	2791	1356	1276
	slowdown	153	272	107
	bounded 10	81	84	60
	bounded 100	23	29	17
	std deviation	80725	83743	71526

Tabela 31 – Métricas para *workload* estática

A média aritmética (*mean*), apesar de confirmar o resultado obtido com performance relativa, pode estar sendo afetada pelos valores extremos da *workload*. Veja que o desvio padrão (*std deviation*) é alto: os dados estão dispersos. O resultado da performance calculado através da média geométrica (*geomean*) nos parece o mais apropriado para agrupar os diversos *turn-around times* em um único valor, pois pelas propriedades já discutidas desta métrica, ela é menos afetada pelos valores extremos da *workload*. Veja que seus resultados são compatíveis com os resultados obtidos com performance relativa. Através destas duas métricas visualizamos que a performance melhora quando alguma das formas de SA é utilizada, e melhora um pouco mais quando SA_i é usado. Entretanto, as métricas baseadas em *slowdown* apresentam resultados contraditórios para os computadores com 500 e 750 processadores. O *slowdown* ($s = tw + te / te$) assume que *te* é fixo. Mas como

temos distintas alternativas de *request*, podemos ter distintos valores de *te*. Torna-se impossível comparar os valores de *slowdown* entre os *requests* estático, com SA_s ou com SA_i, pois provavelmente têm distintos *te*. Mais uma vez nos deparamos com a influência da métrica no resultado da avaliação de performance, pois se a avaliação fosse feita usando apenas métricas baseadas em *slowdown*, as conclusões provavelmente seriam diferentes.

Programa *target* em *workload* em tempo de submissão

Chamamos a *workload* de "em tempo de submissão" quando os programas que compõem a *workload* usam SA_s. Assim como no caso anterior, o programa *target* pode variar da seguinte forma: (i) usar um *request* estático (ii) usar SA_s ou (iii) usar SA_i. A Figura 25 mostra a performance relativa (para o supercomputador com 500 processadores) obtida através da razão entre os *turn-around times* de:

- (ii) *target* e *workload* usando SA_s (sub-sub) / (i) *target* estático em uma *workload* em tempo de submissão (static-sub)
- (i) *target* usando SA_i em *workload* em tempo de submissão (exec-sub) / (i) *target* estático em *workload* em tempo de submissão (static-sub)

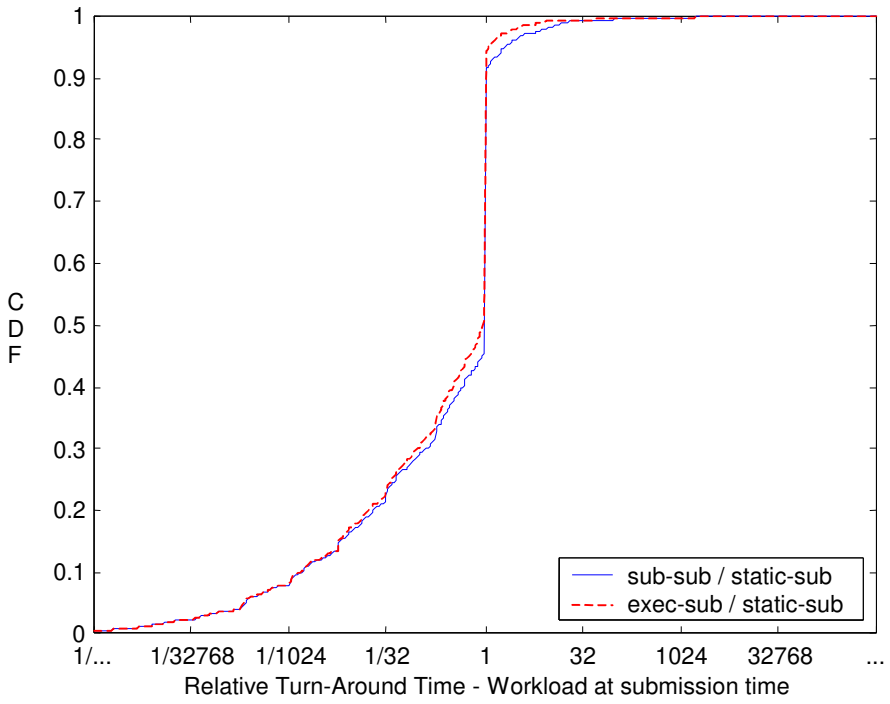

Figura 25 - Performance relativa – *workload* em tempo de submissão

A *workload* está usando SA_s e estamos variando o *target*. No primeiro caso temos o *target* usando o *request* estático comparado com o *target* usando SA_s: quando o *target* usa o *request* estático, fica numa situação muito desfavorável. A performance é bem melhor quando o *target* usa SA_s, competindo com o restante da *workload* em igualdade de condições.

Na segunda situação, temos o *target* usando o *request* estático comparado com o *target* usando SA_i. A performance do *target* usando SA_i é também muito melhor. A diferença entre o uso de SA_s e SA_i não é muito grande, vejamos as Tabelas 31 e 32:

	%
target estático melhor	9.33
igual	44.48
target com SA_s melhor	46.18

Tabela 32 - Sumário de SA_S em *workload* em tempo de submissão

	%
target estático melhor	5.36
igual	44.94
target com SA_i melhor	49.68

Tabela 33 - Sumário de SA_i em *workload* em tempo de submissão

A performance é muito melhor quando o programa *target* usa uma das formas de SA. Entretanto, novamente a diferença de performance entre o uso de SA_s e SA_i é pequena.

Vejamos na Tabela 34 a performance calculada através das diversas métricas discutidas em 3.4.

		target		
	Métrica	Static	SA_s	SA_i
250 procs	*mean*	43612	36001	31493
	geomean	3761	1974	1854
	slowdown	148	163	93
	bounded 10	80	74	54
	bounded 100	32	31	22
	std deviation	127438	114679	102018
500 procs	*mean*	30979	24106	21785
	geomean	2894	1431	1351
	slowdown	98	147	60
	bounded 10	49	49	36
	bounded 100	19	20	16
	std deviation	91713	83646	76339
750 procs	*mean*	28090	22032	19476
	geomean	2617	1219	1145
	slowdown	114	260	114
	bounded 10	59	73	55
	bounded 100	17	21	16
	std deviation	78788	74999	65777

Tabela 34 – Métricas para *workload* em tempo de submissão

As métricas apresentam um comportamento semelhante ao observado na Tabela 31. Quando o *target* usa SA_i é melhor do que quando o *target* usa SA_s, que é melhor do que o *request* estático. Mais uma vez, consideramos a média geométrica como o resultado mais confiável como estatística sumarizadora, pois sofre menos influência dos valores extremos. Através das métricas *slowdown* e *bounded slowdown* temos resultados confusos ocasionados por *te* ser variável. Se a performance fosse avaliada baseada nestas métricas, os resultados seriam enganosos.

Programa *target* em *workload* em tempo de execução

Chamamos a *workload* de "em tempo de execução" quando os programas que compõem a *workload* usam SA_i. Como nos casos anteriores, o programa *target* pode variar da seguinte forma: (i) usar um *request* estático (ii) usar SA_s ou (iii) usar

SA_i. A Figura 26 mostra a performance relativa (para o supercomputador com 500 processadores) obtida através da razão entre os *turn-around times* calculados da seguinte maneira:

- (ii) *target* usando SA_s em uma *workload* em tempo de execução (sub-exec) / (i) *target* estático em uma *workload* em tempo de execução (static-exec)
- (iii) *target* usando SA_i em uma *workload* em tempo de execução (exec-exec) (i) *target* estático em uma *workload* em tempo de execução (static-exec)

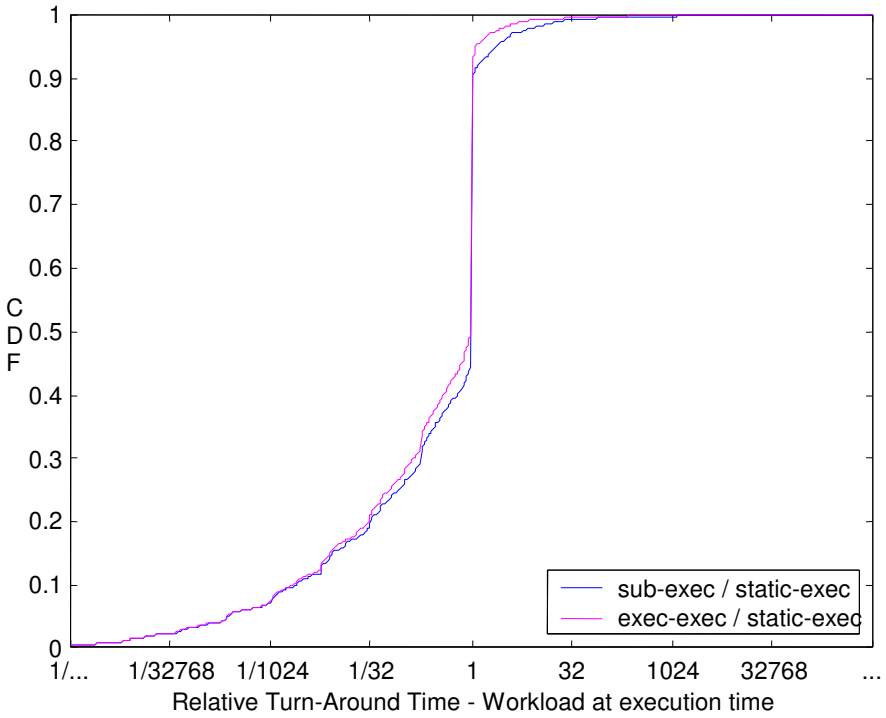

Figura 26 - Performance relativa – *workload* em tempo de execução

Aqui a *workload* decide em tempo de execução. Neste cenário, o *target* obtém melhor performance quando decide em tempo de execução. Vejamos as tabelas 34 e 35:

	%
target estático melhor	9.57
igual	45.40
target com SA_s melhor	45.02

Tabela 35 - Sumário de SA_S em *workload* em tempo de execução

	%
target estático melhor	6.74
igual	42.92
target com SA_i melhor	50.33

Tabela 36 - Sumário de SA_i em *workload* em tempo de execução

Vemos que a performance é muito melhor quando o programa *target* usa SA em qualquer uma das suas formas. Entretanto, a diferença entre o uso de SA_s e SA_i é pequena.

Vejamos na Tabela 37 a performance calculada através das diversas métricas discutidas em 3.4.

		target		
	Métrica	Static	SA_s	SA_i
250 procs	*mean*	41493	32427	30404
	geomean	4268	2320	2200
	slowdown	189	150	115
	bounded 10	89	72	59
	bounded 100	33	29	22
	std deviation	113598	96858	93475
500 procs	*mean*	28073	23633	20974
	geomean	3154	1669	1561
	slowdown	106	176	127
	bounded 10	54	68	54
	bounded 100	21	26	19
	std deviation	72272	72539	63964
750 procs	*mean*	25925	20682	18581
	geomean	2782	1363	1284
	slowdown	98	252	109
	bounded 10	57	74	53
	bounded 100	17	23	15
	std deviation	63636	60745	54622

Tabela 37 – Métricas para *workload* em tempo de execução

O resultado através da média aritmética não nos parece muito representativo, pelo mesmo motivo citado anteriormente: o desvio padrão é muito alto. O resultado com a média geométrica mostra que a performance do programa *target* melhora à medida que usa SA mais próximo do momento da execução, confirmando o que foi visto através da performance relativa. Entretanto, novamente os resultados através das métricas baseadas em *slowdown* não mostram a mesma performance das outras métricas. Pelos mesmos motivos anteriores, não consideramos esta métrica apropriada quando *te* pode ter valores distintos.

Quando todos usam SA

Vimos que o uso do escalonador de aplicação SA melhora a performance da aplicação. Porém, quando SA faz parte do escalonador de recursos (SA_i), a perfor-

mance melhora um pouco mais. Ou seja, olhando do ponto de vista de um programa isolado, SA_i é melhor do que SA_s e ambos são melhores do que usar o *request* estático. Mas o que acontece com a performance quando todos os programas usam SA_s e quando todos os programas usam SA_i? Queremos nesta subseção entender o comportamento emergente do sistema quando todos os programas usam SA. Para tanto, vamos avaliar a performance nos seguintes cenários: (i) a moldabilidade não é usada (ii) a moldabilidade é usada com SA_s (iii) a moldabilidade é usada com SA_i.

Através da Tabela 38 vemos que a moldabilidade com SA_s supera a moldabilidade com SA_i por uma pequena margem para os três supercomputadores simulados.

	static	**SA_s**	**SA_i**
250 procs	3978	1974	2200
500 procs	2999	1431	1561
750 procs	2791	1219	1284

Tabela 38 - Média geométrica para moldabilidade com SA_s e SA_i

Traçamos a performance relativa para entender melhor a relação entre o desempenho obtido pela moldabilidade com SA_s e pela moldabilidade com SA_i. A Tabela 39 e a Figura 27 mostram o resultado obtido para o supercomputador com 500 processadores.

	melhor com SA_s	**igual**	**melhor com SA_i**
%	36.16	27.95	35.89

Tabela 39 - Sumário da performance relativa - SA_s / SA_i

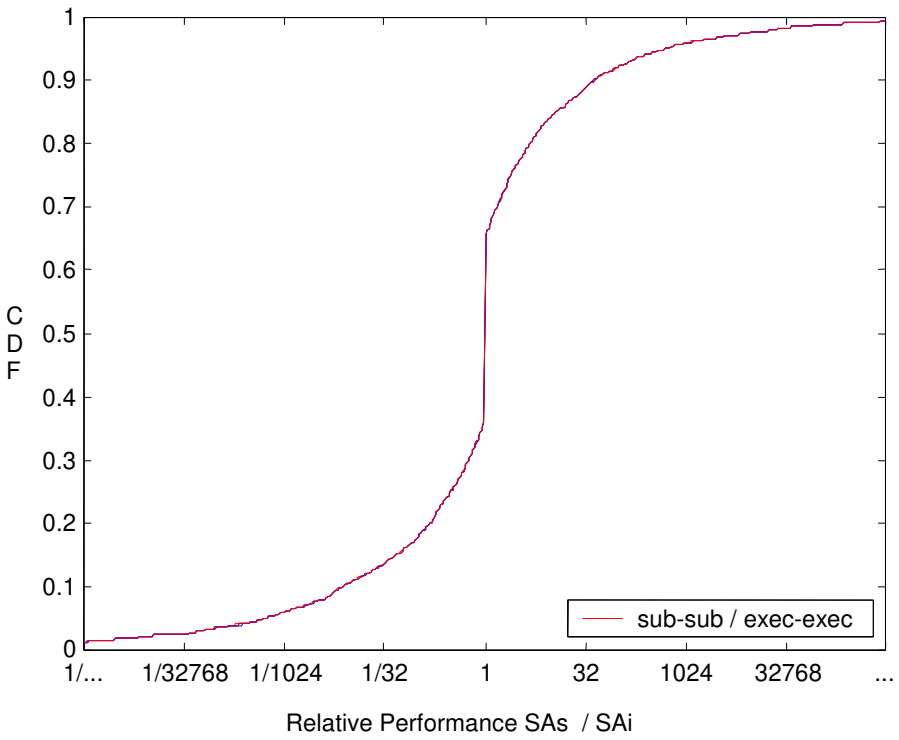

Figura 27 - Performance relativa - SA_s / SA_i

Como podemos ver, quando todos os programas usam SA_i, a performance global do sistema é um pouco pior do que quando todos os programas usam moldabilidade com SA_s. O uso de SA_i demonstra um comportamento egoísta. SA_i melhora a performance da sua aplicação, mas não contribui para melhorar a performance do sistema. Mas, de certa forma, SA_s é egoísta também. Porém, SA_s mostra um comportamento agregado benéfico em algumas situações, como visto em 5.1. Quando a carga aumenta, SA_s escolhe menores *requests*, aumentando a eficiência do sistema, uma vez que a maioria dos programas paralelos tem *speed-up* sublinear, o que reduz a carga oferecida e reduz longos *wait times*. Além disso, um melhor empacotamento dos programas e o fato de existirem menos programas no sistema tornam mais fácil para os programas que chegam na fila de espera serem escalonados, conseqüentemente reduzindo os *wait times*. Em suma, acreditamos que SA_s se adapta à carga oferecida e SA_i não. Como SA_i decide baseado na situação atual do supercomputador e no momento da execução SA_i conhece exatamente quais são os recursos disponíveis, escolhe o *request* que obtém menor *turn-around time*. O *request* que obtém menor *turn-around time* normalmente é aquele que usa mais

processadores. A conseqüência dessa ação é que alguns programas que estão chegando podem não encontram recursos suficientes e acabam tendo um maior tempo de espera (*tw*). Este aumento do tempo de espera aumenta o *turn-around time* destes programas (*tt* = *tw* + *te*), neutralizando os benefícios que SA obtém pela diminuição do tempo de execução, e até piorando a performance global dos programas.

A fim de comprovarmos esta hipótese de que aumentar o número de processadores para o máximo possível poderia piorar a performance global, realizamos um experimento. A partir de um *log* real de uma *workload* estática com 28.474 programas (KTH, descrita em 4.2), aumentamos o número de processadores de todos os *requests* para o máximo de processadores, no caso, 100. Consideramos um *speed-up* linear para calcular o novo tempo de execução. O resultado obtido está na Tabela 40, na Tabela 41 e na Figura 28.

KTH – 100 *nodes*	***request* original**	***request* com *speed-up* linear**
geomean	1628	5834

Tabela 40 - Média geométrica com *speed-up* linear

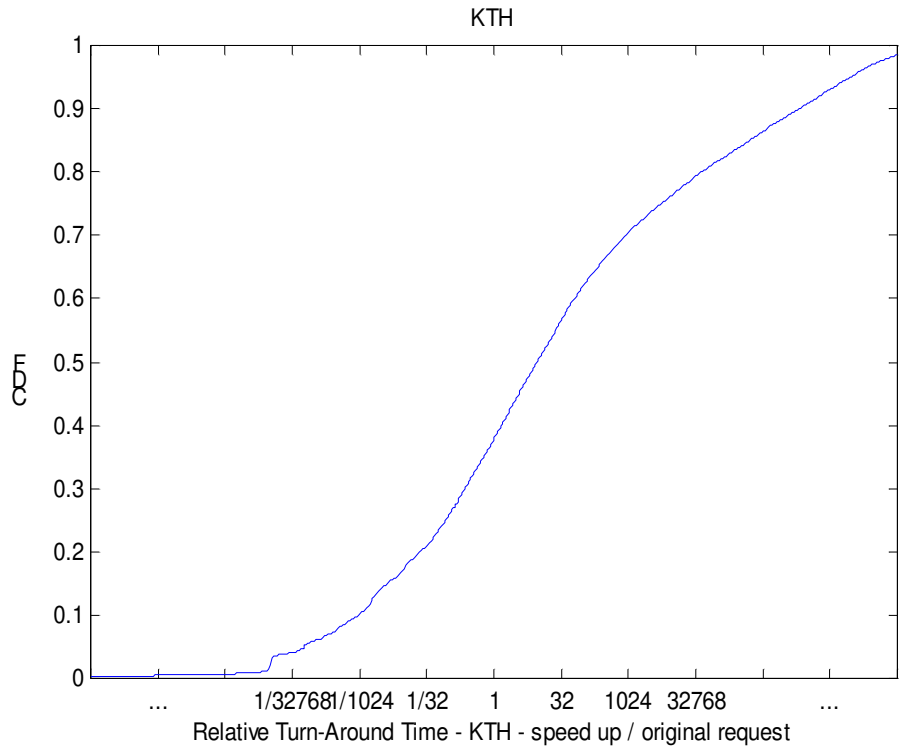

Figura 28 - Performance relativa *speed-up* linear

	melhor com *speed-up* linear	igual	melhor com *request* original
%	37.75	-	62.25

Tabela 41 - Sumário da performance relativa para *speed-up* linear

O que aconteceu foi que cada programa se "apossou" de todos os processadores que podia, e o *turn-around time* subiu enormemente. Apesar do tempo de execução diminuir quando se usa mais processadores, o *wait time* aumenta, porque todos os outros programas estão fazendo o mesmo. Isto redunda em um efeito negativo maior do que o efeito positivo de diminuir o tempo de execução.

Este resultado nos permitiu compreender melhor o que estava acontecendo na nossa simulação. Quando SA_i decide no início da execução do programa tende a escolher o *request* com maior número de processadores. Esta escolha traz o benefício de diminuir o tempo de execução, mas o efeito positivo trazido por esta decisão pode ser anulado pelo aumento do *wait time* dos outros programas que precisam esperar pelos recursos para executar.

5.4. Conclusão

O uso de qualquer uma das variações de SA (SA_s ou SA_i) melhora a performance da aplicação que escalona. Analisamos o comportamento do programa *target* isoladamente, variando desde o uso de *request* estático, usando SA_s e usando SA_i nos seguintes cenários: (i) *workload* estática (ii) *workload* usando SA_s e (iii) *workload* usando SA_i e constatamos que a performance sempre melhora com o uso de SA. A diferença de performance entre o uso de *request* estático e o uso tanto de SA_s como de SA_i é grande, mostrando que o uso de SA realmente beneficia consideravelmente a aplicação. Mais do que isso, vimos que usar SA_i é ainda melhor do que usar SA_s. Entretanto, a melhoria de performance entre o uso de SA_i e o uso de SA_s é pequena. O fato de SA_i ser ainda melhor é explicável porque no momento da execução SA_i pode escolher o *request* baseado na situação real do supercomputador, e assim escolhe o *request* com maior número de processadores, melhorando o tempo de execução e conseqüentemente o *turn-around time* da aplicação. Então, isoladamente, SA_i é melhor do que SA_s. Entretanto, se todo o conjunto de programas da *workload* resolve usar SA_i, o comportamento agregado piora o desempenho global do sistema.

Estamos diante de um problema clássico de cooperação conhecido como *Dilema do Prisioneiro* [2], onde a busca de cada um por seu próprio interesse provoca um resultado pior para todos. No dilema do prisioneiro existem dois jogadores. Na formulação de [2], cada um tem duas alternativas, cooperar ou trair, e cada um deve fazer a sua escolha sem saber o que o outro fará. O jogo está ilustrado na Tabela 42. Um jogador escolhe uma linha, cooperar ou trair. O outro jogador escolhe uma coluna, trair ou cooperar. Juntas, estas escolhas resultam em uma das quatro possibilidades mostradas na tabela. Se ambos os jogadores cooperarem, os dois se saem bem: ambos ganham 3 pontos, a recompensa por cooperação mútua. Se um jogador cooperar mas o outro trair, o jogador que traiu ganha 5 pontos, chamado de tentação para trair e o que cooperou ganha 0 pontos. Se ambos traírem, cada qual ganha 1 ponto, chamado de punição por trair.

jogador da linha	*jogador da coluna*	
	Cooperar	***Trair***
Cooperar	*linha=3, coluna=3*	*linha=0, coluna=5*
Trair	*linha=5, coluna=0*	*linha=1, coluna=1*

Tabela 42 - Dilema do prisioneiro

Quando um jogador vai decidir, raciocina que é melhor trair se o outro jogador cooperar. Mas se o outro jogador trair é melhor que ele traia também, senão não ganha nada. Portanto, qualquer que seja a decisão tomada pelo outro jogador, a melhor alternativa é trair. Só que os dois jogadores raciocinam da mesma forma. Assim, ambos traem. Mas se os dois traem, vão ganhar apenas 1 ponto, que é pior do que os 3 pontos se ambos cooperassem.

Em suma, não importando o que o outro jogador fará, trair trará uma melhor recompensa do que cooperar. O dilema é que se ambos traírem terão uma pior recompensa do que se ambos cooperarem. Portanto, o raciocínio individual leva a um pior resultado para ambos. Aqui está o dilema. A escolha dominante é trair, mas a traição mútua gera um resultado pior para ambos.

Adaptando para o nosso problema, como para todos os cenários o melhor para o programa *target* é usar SA_i, então a escolha dominante é usar SA_i. Por outro lado, quando todos usam SA_i, é pior do que quando todos usam SA_s. Colocando na linguagem do dilema do prisioneiro, quando todos escolhem a melhor opção racionalmente, pensando em seu próprio benefício, provocam uma situação em que o resultado é pior para todos. Veja a Tabela 43.

workload	***target***	
	SA_s	**SA_i**
SA_s	*bom para todos*	*melhor para target, pior para workload*
SA_i	*pior para target, ligeiramente melhor para workload*	*pior para todos*

Tabela 43 - Dilema do prisioneiro para SA_s e SA_i

Embora o usuário não saiba se os outros programas estão usando SA_s, SA_i ou nenhum deles (que é o que atualmente acontece, pois cada usuário submete o seu programa individualmente), a sua decisão está afetando o comportamento dos outros. Se todos resolverem usar SA_i para obter melhor performance, a performance global cairá. Então a decisão que trará um ganho considerável de performance e não prejudicará nem os outros nem a si mesmo é usar SA_s.

Pela própria característica da natureza humana, a escolha dominante é usar SA_i. A maneira de prevenir o uso de SA_i é simplesmente não mudar o escalonador do supercomputador, impossibilitando o uso de SA_i. Assim, através de um controlador central que é o escalonador, o usuário é impedido de tomar a decisão mais racional, restando-lhe escolher entre usar o *request* estático ou usar SA_s. Pelo ganho de performance que SA_s proporciona, aliado ao comportamento emergente, certamente a melhor decisão que o usuário pode tomar é usar SA_s.

Adicionalmente, usamos algumas métricas para avaliar a performance nos diversos cenários. Encontramos resultados contraditórios através das métricas baseadas em *slowdown*. Estes resultados podem ser explicados porque *slowdown* considera que o *te* é fixo. Como para os programas moldáveis dependendo do *request* escolhido para submeter o programa o *te* pode mudar, esta métrica não é recomendada. Além disso, essa métrica pode ser melhorada aumentando *te*. A média aritmética não nos pareceu uma boa maneira de expressar os resultados porque o desvio padrão foi muito alto, evidenciando que os dados estavam muito dispersos. Consideramos que a representação mais adequada foi feita através da média geométrica como estatística sumarizadora, e da performance relativa como uma maneira de representar o desempenho usando toda a distribuição da *workload*.

Capítulo VI

6. Conclusões

A possibilidade de utilizar largamente o processamento paralelo usando Supercomputadores Paralelos (*MPPs*), Redes de estações de trabalhos (*NOWs)* e *Grids* Computacionais, constitui uma alternativa fascinante aos Multiprocessadores Simétricos (*SMPs*) para a busca por resultados computacionais mais rápidos. Esta busca tem sido parte da história da evolução da Ciência da Computação. A cada dia estão sendo descobertas novas aplicações que podem fazer uso desse imenso potencial, além daquelas conhecidas, que demandam muito poder de processamento ou são intensivas em dados. A ampla possibilidade de combinação dos recursos disponíveis nestas plataformas faz com que a nova capacidade de processamento proveniente destas combinações possa ser considerada infinita [16].

Os Supercomputadores Paralelos de Memória Distribuída ou *MPPs* trazem intrínsecos ao seu ambiente características que influenciam determinantemente sobre a sua performance. E uma pequena melhoria de performance representa muito para aplicações que podem demorar dias, semanas ou até meses para serem processadas [45].

Entre as características que influenciam a performance dos supercomputadores paralelos encontra-se a ***Workload***. A influência da *workload* é o aspecto mais amplamente discutido na literatura. A distribuição dos tamanhos dos programas, o horário em que são submetidos e até a área dos *requests* são algumas das características da *workload* que afetam o desempenho dos supercomputadores [17] [22] [36] [37]. Conhecer a carga que está sendo submetida ao supercomputador e poder caracterizá-la possibilita analisar e compreender melhor o comportamento do supercomputador [6] [10] [36] [37]. Caracterizar a *workload* permite construir modelos abrangentes a fim de simular seu processamento e estudar as variações de desempenho do supercomputador quando exposto a determinadas características da carga [10] [14] [17]. Entretanto, a *workload* é apenas um entre os fatores que exercem influência sobre a performance dos supercomputadores.

A **Métrica** é o segundo fator que desempenha um papel importante no processo de avaliação de performance dos supercomputadores paralelos. Curiosamente, a métrica deveria ser apenas uma maneira de representar o desempenho e não poderia afetar a avaliação de performance. Porém não é assim que acontece. Semelhantemente ao que a literatura mostra [20] [21] [25] [42], encontramos nos nossos estudos de caso a influência determinante da métrica. A métrica inadequada pode sofrer influência da *workload* e mostrar resultados confusos e tendenciosos. Entre outras características da influência das métricas sobre o processo de avaliação de performance, encontramos nos nossos estudos de caso:

- A influência das métricas sumarizadoras. A sumarização dos resultados em um número que representa a performance do supercomputador pode ocultar muita informação por trás deste número. Além disso, a sumarização pode estar sendo influenciada pelas características da *workload*, que nos supercomputadores costuma ser muito elástica. Temos como exemplo a média aritmética dos *turn-around times*.
- Resultados conflitantes através das métricas baseadas em *slowdown*. Tanto no estudo de caso sobre o dilema *Easy backfilling* × *Conservative backfilling*, quanto na avaliação de alternativas de escalonamento de programas moldáveis, tivemos resultados conflitantes através destas métricas. O *slowdown* ($s = tw + te / te$) assume que te é fixo. Mas, para programas moldáveis, como temos distintas alternativas de *request*, podemos ter distintos valores de te. Torna-se impossível comparar os valores de *slowdown* entre os *requests* estático e os *requests* que usam moldabilidade, pois provavelmente têm distintos te. Este é um exemplo do uso inadequado de uma determinada métrica de performance. Além disso, para melhorar esta métrica, isto é, para baixar o valor do *slowdown* o sistema poderia fazer com que os programas tivessem um tempo de execução maior.
- A média geométrica dos *turn-around times*, apesar de ser também uma métrica sumarizadora, mostrou ser menos influenciada pelos valores extremos da *workload*.

Este trabalho propõe amenizar o problema da interferência das métricas sobre a avaliação de performance através do uso de **Performance Relativa**. Performance relativa, através da comparação direta dos *turn-around times* dos métodos de escalonamento, minimiza a interferência da métrica sobre o resultado da avaliação de performance através de uma representação dos resultados que é pouco afetada pelos valores extremos. Esta propriedade de ser pouco afetada é conseguida através da comparação dos *turn-around times* programa a programa, abrangendo toda a distribuição da *workload*. Além disso, proporciona uma boa visualização gráfica que dá uma idéia do impacto do método de escalonamento sobre os programas.

Mas talvez a contribuição mais importante desta análise sobre as métricas de performance seja metodológica, apresentando uma métrica que, por não usar estatísticas sumarizadoras, é fundamentalmente diferente das métricas até então utilizadas. Obviamente, ao apresentarmos a métrica performance relativa não pretendemos invalidar as outras métricas anteriormente utilizadas, pois o processo de avaliação de performance não tem como objetivo único concluir quem é melhor. Mais do que isso, o que nos interessa é compreender cada vez melhor o comportamento dos escalonadores ao lidar com determinadas características das *workloads*. Neste sentido, seria indicado conduzir qualquer avaliação de performance usando o maior número de métricas e os mais completos modelos de carga disponíveis, e não apenas um em particular. Se os resultados obtidos forem conflitantes, deve-se analisar detalhadamente o porquê e procurar conhecer as causas dos conflitos, isolando-os e refazendo o processo com os dados e métricas que forem mais significativos para o objetivo do estudo.

O terceiro fator é o **Escalonador**. O escalonador tem o papel de gerenciar a fila de execução de modo a aproveitar da melhor maneira possível os recursos do supercomputador. O método de escalonamento diz respeito à maneira como os programas são alocados para execução. A performance dos supercomputadores está diretamente ligada à atuação do escalonador. Os escalonadores usados atualmente (Easy [35], PBS [32], Maui [40], LSF [46], CRONO [43], entre outros) aceitam apenas requests estáticos.

No nosso estudo de caso sobre o dilema *Easy backfilling* × *Conservative backfilling* [25] [42] realizamos uma análise de performance destes dois métodos de escalonamento e chegamos a algumas conclusões:

- De modo geral, o método de escalonamento *Easy backfilling* proporcionou uma melhor performance para as *workloads* analisadas do que o método *Conservative backfilling*.
- Observamos que o maior impacto individual foi do tempo requisitado (*tr*). Enquanto para os programas com *tr* pequeno *Easy* e *Conservative* são mais semelhantes, *Easy* foi muito melhor para os programas grandes com relação à *tr* do que *Conservative*. Ainda mais, a diferença de performance entre *Easy* e *Conservative* parece aumentar à medida que *tr* aumenta. Este fenômeno pode ser explicado porque ao realizar *backfilling* quando *tr* é grande, *Easy* pode "abrir espaço" para encaixar um programa (desde que não atrase o primeiro da fila), mas *Conservative* não pode.
- Percebemos que, para o número de processadores (*n*) pequeno, o método *Easy* apresentou performance bem melhor. Este fato também pode ser explicado porque *Easy* pode "empurrar" os outros programas para trás (exceto o primeiro da fila), se o *tr* do programa que está sendo encaixado assim o exigir. Dessa forma, *Easy* consegue encaixar mais facilmente os programas com *n* pequeno e *tr* grande. É interessante notar também que o fato de *Easy* beneficiar os programas com *n* pequeno não afeta negativamente os programas que têm *n* médio e grande. A performance de *Easy* é melhor também para os programas médios e grandes com relação à *n*.
- Observamos que nas muitas vezes em que *Easy* "empurra" um programa para trás, este programa acaba não sendo prejudicado, pois aquele programa que o "empurrou" termina muito antes do tempo requisitado. Suspeitamos que isto ocorre porque os tempos requisitados (*tr*) são estimativas ruins dos tempos efetivamente usados na execução dos programas (*te*) [10].

- Por último, verificamos uma sensível diferença entre a curva de performance relativa dos programas pequenos segundo o tempo requisitado (*tr*), a área requisitada ($tr \times n$), o tempo de execução (*te*) e a área de execução ($te \times n$) e a curva de performance relativa dos programas médios e grandes: a curva é mais abrupta para os programas pequenos. Ou seja, a atuação do escalonador tem maior impacto sobre os programas pequenos segundo estas características. Acreditamos que este fenômeno deve-se a dois fatores. Primeiro, programas menores são mais fáceis de fazer *backfilling* e portanto tem mais chance de se beneficiar da atuação do escalonador. Porém, programas com pequeno *te* não são necessariamente mais fáceis de escalonar. Programas com pequeno *te* podem ter grande *tr* e serem mais difíceis de escalonar. Então investigamos uma segunda explicação para o fenômeno e concluímos que a redução do tempo de espera (*tw*) tem mais impacto em programas menores, uma vez que $tt = tw + te$. A manifestação do fenômeno de curvas bruscas também para programas com pequeno *te* indica que esta segunda razão é realmente crucial para explicar os resultados observados.

Estas foram as nossas conclusões quando realizamos a análise de performance dos métodos de escalonamento *Easy* e *Conservative backfilling*, que recebem *requests* estático. Porém, a maioria dos programas paralelos hoje é moldável, isto é, podem ser executados com diferentes *requests* [14]. SA é um escalonador de aplicação que freqüentemente melhora a performance dos programas que escalona [9]. A boa melhoria de performance obtida por SA aliada ao bom comportamento emergente quando muitas instâncias de SA estão atuando no sistema [11] nos levou a projetar uma variação de SA agindo também no momento do início da execução dos programas (que chamamos de SA_i). Esta versão foi implementada como um módulo do escalonador de recursos do sistema. Apesar de SA_i obter melhor performance quando utilizado isoladamente por um programa, quando toda a *workload* está usando SA_i a performance global piora. Este é um problema clássico de cooperação, conhecido como o *Dilema do Prisioneiro* [2]. No Dilema do Prisioneiro, a busca de cada um por seu próprio interesse provoca um resultado pior para todos. Isto é, a escolha dominante tomada por cada programa pretendendo obter a máxima performance possível usando SA_i piora o desempenho

máxima performance possível usando SA_i piora o desempenho de todos. A solução que beneficia a todos sem prejudicar a performance global é todos usarem a versão original de SA. Porém, pela tendência natural de querer sempre o máximo de vantagem, a escolha dominante do usuário será usar SA_i. Deste modo, a solução para que cada usuário não venha a escolher usar SA_i e assim possa a prejudicar o desempenho de todos, inclusive o dele, é não mudar o escalonador do sistema. Não havendo a implementação de SA_i no escalonador de recursos do sistema, a melhor opção possível para o usuário será utilizar a versão original de SA. Como já vimos, esta opção é bastante satisfatória pela otimização de performance que proporciona com relação ao uso de *request* estático e também pelo comportamento emergente das muitas instâncias de SA atuando no supercomputador. O comportamento coletivo ganha quando se escolhe usar SA.

A avaliação de performance dos supercomputadores paralelos envolve a w*orkload*, as métricas de performance e o escalonador. É praticamente impossível analisar estes fatores de forma independente, pois se encontram intrinsecamente ligados. Além disso, a interação entre eles introduz novas características que influenciam a performance dos supercomputadores paralelos, tornando o processo de avaliação de performance bem mais complexo. Concluímos afirmando que gerar resultados de avaliações cada vez melhores e que representem da forma mais clara e compreensível possível o desempenho dos supercomputadores paralelos nos levará a processos de avaliação de performance completos, esclarecedores e livres de conclusões distorcidas e enganosas.

Referências Bibliográficas

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the *NOW* Team. ***A Case for Networks of Workstations: NOW.*** IEEE Micro, Feb, 1995.
http://now.cs.berkeley.edu/Case/case.ps

[2] R. Axelrod**.** ***The Evolution of Cooperation***. Basic Books, 1984.

[3] D. H. Bailey, ***Misleading Performance Reporting in the Supercomputing Field,*** Scientific Programming, vol. 1., no. 2 (Winter 1992), pg.141-151.
http://www.nersc.gov/~dhbailey/dhbpapers/mislead.ps

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. ***Application-Level Scheduling on Distributed Heterogeneous Networks***. Proceedings of Supercomputing'96.
http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html

[5] F. Berman and R. Wolski. ***The AppLeS Project: A Status Report.*** In Proceedings of the 8th NEC Research Symposium, Berlin, Germany, May 1997.
http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html

[6] M. Calzarossa and G. Serazzi. ***Workload Characterization: A Survey***. Proceedings of the IEEE, vol. 81, no. 8, pp. 1136-1150, Aug 1993.
http://mafalda.unipv.it/Laboratory/publications/PS/IEEE93.ps.gz

[7] M. Calzarrossa and R. Marie. ***Tools for Performance Evaluation.*** **Performance Evaluation** – An International Journal, 33(1):1-3, 1998.
http://mafalda.unipv.it/Laboratory/publications/PDF/Special_issue.pdf

[8] S. Chapin, W. Cirne, D. Feitelson, J. Jones, S. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. ***Benchmarks and Standards for the Evaluation of Parallel Job Schedulers.*** In Job Scheduling Strategies

for Parallel Processing. D. Feitelson and Larry Rudolph (Eds.) Springer-Verlag, Lecture Notes in Computer Science, vol. 1659, pp. 66-89, 1999.
http://walfredo.dsc.ufcg.edu.br/papers/parallel-sched-bench.ps

[9] W. Cirne. ***Using Moldability to Improve the Performance of Supercomputer Jobs***. PhD Thesis, University of California, San Diego. 2001.
http://walfredo.lsd.ufcg.edu.br/papers/thesis.pdf

[10] W. Cirne and F. Berman. ***A Comprehensive Model of the Supercomputer Workload.*** In Proceedings of WWC-4: IEEE 4th Annual Workshop on Workload Characterization, Dec 2001.
http://walfredo.dsc.ufcg.edu.br/papers/comprehensive-model.ps

[11] W. Cirne and F. Berman. ***When the Herd is Smart: Aggregate Behavior in the Selection of Job Request.*** In IEEE Transactions in Parallel and Distributed Systems, vol. 14, no. 2, pp 181-192, Feb 2003.
http://walfredo.lsd.ufcg.edu.br/papers/when-the-herd-is-smart.pdf

[12] W. Cirne. ***Grids Computacionais: Arquiteturas, Tecnologias e Aplicações.*** In Anais do Terceiro Workshop em Sistemas Computacionais de Alto Desempenho, Vitória – ES, Out 2002.
http://walfredo.dsc.ufcg.edu.br/papers/Grids%20Computacionais-%20WSCAD.pdf

[13] W. Cirne. ***MyGrid Project.***
http://www.dsc.ufcg.edu.br/mygrid/

[14] W. Cirne and F. Berman. ***A Model for Moldable Supercomputer Jobs.*** Proceedings of the IPDPS 2001 – International Parallel and Distributed Processing Symposium, San Francisco, USA, Apr 2001.
http://walfredo.lsd.ufcg.edu.br/papers/moldability-model.pdf

[15] P. L. Costa Neto. ***Estatística***. Editora Edgard Blücher Ltda, 1977.

[16] J. Dongarra. ***WWW + Grid Computing = Next Generation Web.*** In Football PreGame Showcase, University of Tenessee, Nov, 2000.

http://www.netlib.org/utk/people/JackDongarra/SLIDES/football-showcase-2000.pdf

[17] A. Downey and *D.* Feitelson. ***The Elusive Goal of Workload Characterization.*** Perf. Eval. Rev. 26(4), pp. 14-29, March 1999.

http://www.cs.huji.ac.il/~feit/papers/Elusive99PER.ps.gz

[18] A. Downey. ***A Parallel Workload Model and its Implications for Processor Allocation.*** In 6th Intl. Symp. High Perf. Distributed Computing, Aug 1997.

http://www.sdsc.edu/~downey/allocation/

[19] A. Downey. ***Predicting queue times on space-sharing parallel computers.*** 11th International Parallel Processing Symposium (IPPS'97), Geneva, Switzerland, Apr 1997.

http://www.sdsc.edu/~downey/predicting/

[20] D. Feitelson. ***Analyzing the Root Causes of Performance Evaluation Results.*** Technical Report 2002-4, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Mar 2002.

http://www.cs.huji.ac.il/~feit/papers/Root02TR.ps.gz

[21] D. Feitelson. ***The Effect of Metrics and Workloads on Performance Evaluation.*** Technical Report 2001-13, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Oct 2001.

http://www.cs.huji.ac.il/~feit/papers/design.ps.gz

[22] D. Feitelson. ***Sensitivity of Parallel Job Scheduling to Fat-Tailed Distributions.*** October 2001. Technical Report 200-44 Inst. Computer Science, The Hebrew University of Jerusalem, Oct 2000.

http://leibniz.cs.huji.ac.il/*tr*/acc/2000/HUJI-CSE-LTR-2000-44_disp.ps.gz .

[23] D. Feitelson**.** ***Metrics for Parallel Job Scheduling and Their Convergence.*** In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), pp. 188-206, Springer-Verlag, 2001. Lecture Notes in Computer Science Vol. 2221. © Copyright2001 by Springer-Verlag.

http://link.springer.de/link/service/series/0558/bibs/2221/22210188.htm

[24] D. Feitelson and L. Rudolph. ***Metrics and Benchmarking for Parallel Job Scheduling.*** In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), pp. 1-24, Springer-Verlag, Lecture Notes in Computer Science vol. 1459, 1998.
http://www.cs.huji.ac.il/~feit/parsched/p-98-1.ps.gz

[25] D. Feitelson and A. Mu'Alem. ***Utilization and Predictability in Schedulling IBM SP2 with Backfilling.*** In 12th Intl. Parallel Processing Symposium, pp 542-546, Apr 1998.

[26] D. Feitelson, L. Rudolph, U. Schweigelshohn, K. Sevcik, and P. Wong. ***Theory and Practice in Parallel Job Scheduling***. In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), pp. 1-34, Springer-Verlag, 1997. Lecture Notes in Computer Science Vol. 1291.
http://www.cs.huji.ac.il/~feit/parsched/p-97-1.ps.gz

[27] D. Feitelson, ***Packing Schemes for Gang Scheduling.*** In Job Scheduling Strategies for Parallel Processing, pp. 89-110, Springer-Verlag, 1996. Lecture Notes Comput. Sci. vol. 1162.
http://www.cs.huji.ac.il/~feit/parsched/p-96-6.ps.gz

[28] I. Foster. ***Designing and Building Parallel Programs.*** *Livro On-line.*
http://www-unix.mcs.anl.gov/dbpp/

[29] I. Foster and C. Kesselman. ***The Grid Blueprint for a New Computing Infrastructure.*** Morgan Kauffman, 1998.

[30] R. Gibbons. ***A Historical Application Profiler for Use by Parallel Schedulers.*** Lecture Notes in Computer Science, vol. 1297, 58-75, Springer-Verlag, 1997.

[31] V. Hazlewood. ***NPACI JobLog Repository***.
http://joblog.npaci.edu/

[32] R. Henderson. ***Job Scheduling Under the Portable Batch System.*** In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Lecture Notes in Computer Science Vol. 949, pp. 337-360, Springer-Verlag, 1995.

[33] R. Jain. ***The Art of Computer System Performance Analysis***. Wiley, 1991.

[34] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan. ***Modeling of Workload in MPPs.*** In Job Scheduling Strategies for Parallel Processing, pp. 95-116, Springer-Verlag, 1997. Lecture Notes Comput. Sci. vol. 1291.
http://www.cs.huji.ac.il/~feit/parsched/parsched97.html

[35] D. Lifka. ***The ANL/IBM SP Scheduling System.*** In IPPS 1995 - Workshop on Job Scheduling Strategies for Parallel Processing, Apr 1995.
http://www.tc.cornell.edu/~lifka/Papers/ipps95.pdf

[36] V. Lo, J. Mache, and K. Windisch. ***A Comparative Study of Real Workload Traces and Synthetic Workload Models for Parallel Job Scheduling***. In Job Scheduling Strategies for Parallel Processing, Dror Feitelson and Larry Rudolph (Eds.), Springer Verlag, Lect. Notes Comput. Sci. vol. 1459, pp. 25-46, 1998.
http://www.cs.huji.ac.il/~feit/parsched/parsched98.html

[37] U. Lublin and D. G. Feitelson, ***The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs***. In Technical Report 2001-12, Hebrew University, Oct 2001.
http://www.cs.huji.ac.il/labs/parallel/publications.shtml

[38] Y. Mahéo, L. Massari and P. Rossaro. ***Performance Evaluation of Automatically Generated Data-Parallel Programs***. In Proc. 4th EuroMicro Workshop on Parallel and Distributed Processing, pages 534-540, 1996.
http://mafalda.unipv.it/Laboratory/publications/PDF/EuroMicro96.pdf

[39] L. Massari and P. Rossaro. ***Workload Characterization Methodologies in Parallel Environment***. In Proc. Seventh International Symposium on Computer and Information Sciences (ISCIS), pages 519-522, Nov 1992.

http://mafalda.unipv.it/Laboratory/publications/PS/Iscis92.ps.gz

[40] Maui High Performance Computing Center. ***The Maui Scheduler Web Page.***

http://mauischeduler.sourceforge.net/

[41] J. A. Moreira and *W*. Cirne. ***Usando Performance Relativa para Avaliar Desempenho em Supercomputadores Paralelos.*** In Anais do III Workshop em Sistemas Computacionais de Alto Desempenho, Vitória-ES, Out 2002.

http://walfredo.lsd.ufcg.edu.br/papers/perf_relativa.ps

[42] A. Mu'Alem and D. Feitelson. ***Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling.*** IEEE Trans. Parallel & Distributed Syst, 12(6), pp. 529-543, Jun 2001.

http://www.cs.huji.ac.il/~feit/papers/SP2backfil01TPDS.ps.gz

[43] M. A. S. Netto and C. A. F. De Rose, CRONO: ***A Configurable Management System for Linux Clusters.*** The Third LCI International Conference on Linux Clusters: The HPC Revolution 2002 (LCI'2002), 2002.

http://www.democritos.it/activities/IT-MC/cluster_revolution_2002/PDF/21-DeRose_C.pdf

[44] ***Parallel Workloads Archive.***

http://www.cs.huji.ac.il/labs/parallel/workload

[45] K. T. Pedretti, Th. L. Casavant, R. C. Braun, T. E. Scheetz, C. L. Birkett, C. A. Roberts. ***Three Complementary Approaches to Parallelization of Local BLAST Service on Workstation Clusters.*** Parallel Computing Technologies, pp. 271-282, 1999.

http://www.eng.uiowa.edu/~pedretti/stuff/blast/paper.ps

[46] Platform Computing Corp. ***Load Sharing Facility Web Page.*** http://www.platform.com/products/wm/LSF/index.asp

[47] D. Ridge, D. Becker, P. Merkey, T. Sterling. ***Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs.*** Proceedings of IEEE Aerospace, 1997. http:/www.beowulf.org/paper.ps/papers.html

[48] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, Mei-Hui Su, C. Kesselman, S. Young, and M. Ellisman. ***Combinig Workstations and Supercomputers to Support Grid Applications: The Parallel Tomography Experience.*** Proceedings of the HCW'2000 - Heterogeneous Computing Workshop, May 2000. http://walfredo.lsd.ufcg.edu.br/papers/tomography.pdf

[49] W. Smith, I. Foster, and V. Taylor. ***Predicting Application Run Times Using Historical Information.*** Lecture Notes in Computer Science, 1459:122-142, Spring-Verlag, 1998. http://www-fp.mcs.anl.gov/~foster/papers.html

[50] W. Smith, V. Taylor, and I. Foster. ***Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance.*** In Proceedings of the IPPS/SPDP'99 Workshop on Job Scheduling Strategies for Parallel Processing, 1999. http://www-fp.mcs.anl.gov/~foster/papers.html

[51] Standard Performance Evaluation Corporation. ***The SPEC web page.*** http://www.spec.org/

[52] W. J. Stevenson. ***Estatística Aplicada à Administração***. Editora Harbra, 1986.

[53] D. Zotkin and P. J. Keleher. ***Job-lenght estimation and performance in backfilling schedulers.*** In 8th High Performance Distributed Computing Conf., 1999. http://motefs.cs.umd.edu/papers/index.pl