Depurando Desempenho de Aplicações Software DSM com Carnival

Edison Ishikawa[1 3], Elcio José Pineschi[1], Wagner Meira Jr[2] e Cláudio Amorim[1]

[1] COPPE - Programa de Engenharia de Sistemas e Computação, UFRJ
Rio de Janeiro, RJ, Brasil, CEP 21945-970
{edsoni, pineschi, amorim }@cos.ufrj.br
[2] Departamento de Ciência da Computação, UFMG
Belo Horizonte, MG, Brasil
{meira@dcc.ufmg.br}
[3] Instituto Militar de Engenharia - IME
Rio de Janeiro, RJ, Brasil

Abstract—
The difficulty of parallel applications to attain reasonable performance levels has motivated the development of performance debugging tools such as Paradyn, StormWatch, and Carnival. Although, developers have usually demonstrated tool capabilities on different kind of applications, often no evaluation of tool effectiveness has been reported from less specialized users facing the task of developing efficient parallel applications with the only support of performance debugging tools. This paper begins to fill this gap. More specifically, we evaluate the effectiveness of Carnival to support the development of software DSM applications by programmers having little background on parallel programming and performance debugging tools. Using Carnival we port four parallel applications to TreadMarks, a well-known software DSM system. Two applications (*Raytrace* and LU) are from the SPLASH-2 suite and other two applications (TSP and IS) belong to the TreadMarks application benchmarks. Our results show that the causes of synchronization and communication problems in the applications were precisely diagnosed by Carnival. In addition, Carnival diagnostics enabled simple code modifications that increased application performance by up to a factor of 2 in TreadMarks applications and as much as 46 times in Splash2 applications. The main difficulty we faced was Carnival large trace files specially in Raytrace which could be solved by augmenting the granularity of the application instrumentation. In overall, Carnival offers a simple and friendly interface for performance vizualization. Our conclusion is that Carnival offers an effective tool to implement efficient software DSM applications.

Keywords— **parallel programming, software DSM, performance analyser**

I. INTRODUÇÃO

O crescente aumento de banda passante das redes de computadores e do poder de processamento dos PCs tornou possível a construção de sistemas baseados em *clusters* de computadores com desempenho equivalente a de supercomputadores a uma fração de o seu custo. É então previsto uma crescente utilização desses sistemas em universidades e empresas, estimulando o desenvolvimento de aplicações e investimento em programadores capazes de extrair o máximo de poder computacional oferecida por estas arquiteturas. Entretanto, a programação paralela é nada trivial pois transfere para o programador a complexa tarefa de orquestrar eficientemente múltiplos processadores que compartilham dados (físicamente distribuídos) através de uma rede de comunicação de alto desempenho mas de alta latência. Isso exige não só programadores que tenham domínio de programação paralela mas também ferramentas de suporte ao desenvolvimento de programação eficiente que permitam aumentar a produtividade desta, considerando que a vasta maioria dos dos programadores não possuêm nenhuma experiência prévia com programação paralela.

A forma tradicional de se desenvolver aplicações paralelas em *clusters* é utilizar o modelo de programação de passagem de mensagens. Este modelo requer alta especialização do programador, que é obrigado a se preocupar com toda a distribuição de dados da aplicação e com toda a comunicação de dados. Em contraste, o modelo de programação de memória compartilhada distribuída (DSM), intuitivamente mais simples, oferece a abstração de uma memória compartilhada sobre um *hardware* de memória distribuída e pode ser inteiramente implementado em *software*, sendo denominado *software* DSM [NL91], ou simplesmente SDSM.

Num sistema SDSM, o programador realiza operações convencionais de escrita e leitura à memória compartilhada sob a gerência do protocolo DSM que se encarrega de manter a coerência dos dados de forma transparente para o programador. Apesar do modelo de programação ser mais conveniente, os sistemas SDSM ainda possuem altos *overheads* de comunicação e sincronização. A depuração de desempenho se agrava pelo fato de ser extremamente difícil para um programador entender a causa do baixo desempenho de sua aplicação nesses sistemas porque protocolos SDSM implementam a gerência de memória, incluindo toda distribuição e comunicação de dados, sem que o programador possa interferir diretamente.

Carnival[MJ97] é uma ferramenta de análise e visualização de desempenho que se propõe a entender o desempenho das aplicações, direcionando o programador para pontos específicos no código fonte do programa responsáveis

pela degradação de desempenho sofrida pela aplicação. *Carnival* se utiliza basicamente de duas técnicas: *waiting time analysis* [MLP96] (WT) e *communication analysis* [MLHA97] (CA). A análise WT visa explicar as causas de tempo de espera sofridas pela aplicação e a análise CA correlaciona os eventos de comunicação de dados causados pelos acessos a dados compartilhados.

O objetivo deste trabalho é de utilizar *Carnival* como ferramenta de referência, e avaliar sua eficácia no processo de depuração de desempenho de aplicações paralelas para sistemas SDSM. Esse processo foi conduzido por programadores com noções básicas de programação paralela mas sem nenhuma experiência prévia com a ferramenta ou com as aplicações paralelas utilizadas. Com isso tenta-se medir a quantidade e a qualidade das informações providas por *Carnival* ao programador para que este possa rapidamente localizar e corrigir os pontos no programa que mais afetam o desempenho da aplicação. É importante também que através dessas informações, o programador tenha pistas de como reestruturar sua aplicação de maneira simples e direta, de forma a aumentar seu desempenho sem que tenha que recorrer a um especialista nem da aplicação e/ou em programação paralela. Convém observar que não é objetivo deste trabalho fornecer informações ao programador para modificar o modelo de consistência de memória do SDSM empregado, mas tão somente depurar o desempenho de aplicações dado que ele irá usar um SDSM que implemente um modelo de consistência de memória qualquer..

A contribuição deste trabalho é avaliar se os problemas de desempenho das aplicações diagnosticados por *Carnival* são suficientes para um programador pouco familiarizado com programação paralela e/ou com a aplicação, conseguir obter desempenho satisfatório de sistemas SDSM. Mais específicamente, sem alterar radicalmente o algoritmo paralelo mas apenas as estruturas de sincronização e distribuição de dados, que grau de desempenho um programador com o suporte oferecido por *Carnival* conseguirá obter de aplicações SDSM?

Nossos resultados preliminares mostram que: 1) a tarefa de instrumentação da aplicação é rápida e objetiva, gerando os traces necessários para análises de desempenho. 2) As análises de *Carnival* indicam precisamente as causas de sincronização e comunicação da aplicação. 3) Estas indicações levaram a modificações no código, na maioria das vezes pequenas, que proporcionaram um aumento significativo para as aplicações que originalmente exibiam *speedups* baixos. Esses resultados preliminares revelam que *Carnival* é uma ferramenta eficaz para depuração de desempenho de aplicações *software* DSM.

Na próxima seção descrevemos a metodologia experimental deste trabalho. Na seção III mostramos o conjunto de aplicações executadas bem com a análise do processo de depuração de desempenho destas aplicações baseda em *Carnival*. Na seção IV apresentamos nossas conclusões e futuros trabalhos.

II. Metodologia Experimental

Para executar os programas paralelos usou-se um IBM SP/2 com seis processadores cada um com 128 MB de RAM interconectados com uma *switch* de 40 MBytes/s, *full-duplex*. *Carnival* utiliza o relógio global ATC da *switch* do SP2 para gravar os *timestamps* nos traces gerados.

Carnival

Carnival é constituída de dois componentes principais:

- *Instrumentação* - Compreende as ferramentas e bibliotecas usadas para adquirir informações sobre a aplicação e sua execução.
- *Análise* - Consiste de ferramentas que analisam os arquivos de traces, produzem a execução de *profiles* e automaticamente geram uma interface gráfica que permite a visualização dos mesmos.

A instrumentação de uma aplicação com *Carnival* mostrou-se uma tarefa simples, necessitando apenas a inserção de comentários no código do programa. Estes comentários seguem uma sintaxe fácil e devem ser colocados em pontos chaves da aplicação como por exemplo procedimentos, barreiras, *locks* e *loops*. Desta forma *Carnival* simplifica sobremaneira esta tarefa, que muitas vezes requer uma quantidade considerável de tentativas e erros para se chegar a um equilíbrio entre a quantidade de instrumentação e o valor das informações que este irá prover[Lar93].

O código assim instrumentado é pré-pro-cessado gerando um novo código acrescido de chamadas ao *runtime system* de *Carnival* que irá gerar o trace, bem como arquivos com informações estáticas sobre o código da aplicação.

Uma vez de posse dos traces, os mesmos são tratados pelo componente de análise de *Carnival*, que gera um código em Tcl/Tk responsável por fazer as visualizações dos diversos *profiles* de execução:

- *Waiting Time Map* - Provê uma perspectiva global de todas as causas de tempo de espera da aplicação ordenada segundo sua importância (percentagem em relação ao tempo total de espera da aplicação).
- *Charac Map* - Apresenta uma explicação para uma única fonte de tempo de espera.

Através destes *profiles* o programador é capaz de observar problemas do tipo: desbalanceamento de carga, desbalanceamento de sincronização, falta de paralelismo, contenção por comunicação, *overhead* do protocolo de consistência de memória, tempo de espera por páginas, etc. Por falta de espaço, não serão apresentadas essas visualizações para as aplicações testadas mas que podem ser encontradas em (http://www.cos.ufrj.br/~pineschi/carnival/sbac99.html).

TreadMarks

TreadMarks[KCDZ94] é um SDSM que implementa o modelo de consistência de memória *lazy release consistency* [KCZ92] e um protocolo do tipo múltiplos escritores baseado em invalidações. *TreadMarks* usa as interfaces e compiladores padrão do UNIX, e utiliza o protocolo UDP/IP para comunicação.

A API do *TreadMarks* é simples e pequena, com facilidades para criação e destruição de processos, sincronização e alocação de memória compartilhada.

Aplicações

Para avaliar *Carnival* como uma ferramenta de depuração de desempenho escolheu-se um pequeno conjunto de aplicações que pudesse cobrir diferentes características da comunicação e sincronização representativas de aplicações paralelas.

Raytrace é uma aplicação que gera uma visualização de uma cena tridimensional usando o algoritmo de *ray trace*[SGL94]. Seu padrão de acesso a dados é altamente irregular e imprevisível. Faz parte da *suite* SPLASH-2 e é um exemplo de uma aplicação migrada de uma arquitetura *hardware* DSM para *software* DSM.

LU é uma aplicação que fatora uma matriz densa em um produto de uma matriz triangular inferior e superior. Ela é uma aplicação cujo acesso aos dados é altamente regular e previsível, mas gera uma grande comunicação entre os nós. Também faz parte da *suite* SPLASH-2 e é um exemplo de porte de aplicação como *Raytrace*.

TSP implementa o problema do caixeiro viajante, em que se busca o menor ciclo hamiltoniano de um grafo. A solução implementada busca a solução ótima através de uma busca exaustiva, porém evitando caminhos desnecessários (*branch-and-bound*). A aplicação acompanha a distribuição do *TreadMarks* e é um exemplo de código já otimizado para o uso de *software* DSM.

A aplicação *IS* (Integer Sort) ordena uma seqüência de chaves usando *bucket sort*. Esta aplicação faz parte da distribuição de *TreadMarks*. IS é uma aplicação com padrão de acesso regular, porém com uma enorme movimentação de dados. A entrada usada foi de 2M chaves representando inteiros na faixa de 0 a $2^{32} - 1$.

A. Metodologia

Como ponto de partida, usamos as duas aplicações (*Raytrace* e LU) da *suite* SPLASH-2. Como estas aplicações não foram desenvolvidas para *software* DSM, houve a necessidade de portá-las primeiro para o ambiente *TreadMarks*.

Outra classe de aplicações que testamos são aquelas que acompanham a distribuição de *TreadMarks*. Selecionamos duas aplicações (TSP e IS) para verificar a eficácia de *Carnival* em aplicações já otimizadas para um ambiente SDSM.

A curva de *speedup* inicial das aplicações foi obtida calculando a razão entre os tempos seqüencial e paralelo gerados variando-se o número de processadores, para cada uma das aplicações do conjunto.

Em seguida, cada aplicação foi instrumentada e executada gerando traces que foram tratados por *Carnival* que gerou as diversas visualizações para análise de desempenho. A partir da identificação do gargalo de desempenho mais relevante, a aplicação é reestruturada com base nas informações fornecidas por *Carnival*. Executa-se novamente a aplicação sem a instrumentação para verificar se houve um aumento no *speedup* da aplicação. Este procedimento é repetido até que não se consiga melhorar mais o código da aplicação, sem ter que reescrever completamente a aplicação. A eficácia de *Carnival* pode ser então avaliada através do aumento proporcionado no desempenho da aplicação sem ter que reescrevê-la completamente.

III. ANÁLISE DAS APLICAÇÕES

Nesta seção apresentamos os resultados obtidos com *Carnival* para cada aplicação instrumentada. O que *Carnival* sinalizou para o programador como sendo o ponto que deveria ser atacado para se obter uma melhora de desempenho, o que o programador fez para atacar o problema, qual a dificuldade para fazê-lo e o que isto refletiu em termos de *speedup*.

A. Ray Trace

A paralelização de *raytrace* foi implementada utilizando-se o modelo de fila de tarefas. A imagem a ser renderizada é dividida em tarefas, onde cada tarefa representa um bloco de pontos desta imagem. Cada processador possui uma fila de tarefas, que contém as tarefas que o mesmo deve realizar. Quando um processador termina todas as suas tarefas ele tenta roubar tarefas da fila de outros processadores no intuito de se obter um melhor balanceamento de carga.

Os *speedups* da versão inicial se encontram na curva 1 da figura 1.

Carnival mostra através do *Waiting Time Map* que mais de 30% do tempo de espera total foi atribuído a uma operação de *lock acquire*, A explicação deste tempo de espera é dada por uma contenção no próprio *lock*, conforme indicam *Charac Map* e o histograma deste escopo. Observando-se o código do programa, vemos que este *lock* tem propósitos apenas estatísticos [JSS97] e que simplesmente removendo o *lock* do programa, este passou a apresentar *speedups* conforme a curva 2 da figura 1.

Em seguida, verificamos que a barreira colocada após a inicialização do programa tornou a parte seqüencial mais importante do que a parte paralela da aplicação. Examinando a parte paralela do código, verificamos que o segundo escopo de maior importância para tempo de espera é a operação de

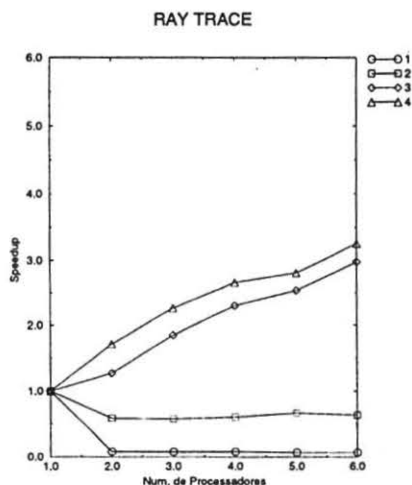

Fig. 1. Speedups obtidos com a depuração de desempenho de Raytrace

acquire no *lock* da função *GlobalMalloc*, e que este *lock* é utilizado para o gerenciamento de memória global realizado na própria aplicação. A inicialização da fila de tarefas que aparentemente era realizada em paralelo pelos processadores estava sendo na verdade serializada por este *lock*. Nossa solução foi implementar a fila de tarefas através de um vetor, evitando a necessidade de alocação dinâmica de memória compartilhada. Os *speedups* para esta versão estão na curva 3 da figura 1.

Esta nova vizualização indica que o tempo de espera na barreira após a inicialização é responsável por mais de 90% do tempo de espera total da aplicação. O próximo gargalo de sincronização da parte paralela da aplicação é o *lock* que guarda as filas de tarefas. Reestruturamos o código da aplicação para que toda vez que um processador tiver que retirar uma tarefa de outro, retirar logo a metade em vez de apenas uma tarefa da fila. Após essa modificação, os *speedups* foram como indicados na curva 4 da figura 1.

B. LU

A primeira versão de LU testada foi a versão não contígua, na qual a matriz a ser fatorada é armazenada num vetor 2D. O desempenho dessa aplicação é dependente da granularidade de comunicação/computação estabelecida que depende do tamanho dos blocos em que a matriz for subdividida. Iniciamos com blocos de 1024x1024 em uma matriz de 1024x1024 com 4 processadores obtendo-se um pequeno *slowdown* de 0,988. *Carnival* identificou imediatamente o problema como sendo de desbalanceamento de carga devido ao excesso de processamento em um dos nós. Para balancear a carga, diminuimos o tamanho dos blocos para 512x512 produzindo-se um modesto *speedup* de 1,69. *Carnival* então sinalizou que o problema continuava a ser desbalanceamento de carga, mas agora devido não só ao desbalanceamento do processamento (66,44%) mas também devido à alta comunicação por causa de falha de páginas(43,55%). Ao diminuir então o tamanho do bloco para 256x256 diminuindo-se o *speedup* para 1,57, refletindo a menor granularidade da aplicação, aumentando o falso compartilhamento. *Carnival* também sinaliza que as falhas de páginas ocorrem nos loops existentes nas funções *bmod* que acessam intensivamente a matriz a ser fatorada.

Portamos então a versão contínua de LU que armazena a matriz a ser fatorada em um vetor 4D. Subdividiu-se a matriz em 4 blocos e *Carnival* indicou que havia um desbalanceamento de carga, apesar do melhor *speedup* alcançado por essa versão. Foi repetido o procedimento anterior até se chegar ao bloco ideal de 64x64.

C. TSP

A versão paralela de TSP aloca em uma fila trechos iniciais de um caminho que pode vir a ser um percurso promissor. Os processos retiram os caminhos da fila e recursivamente testam os percursos restantes para achar um ciclo hamiltoniano mais curto. *Carnival* mostrou que a maior causa de tempo de espera na aplicação era a última barreira do código, e que isto era devido ao paralelismo insuficiente do procedimento recursivo de visita aos nós do grafo. De fato, a visita recursiva aos restante dos nós do grafo à procura de um ciclo hamiltoniano é uma busca exaustiva e de duração imprevisível, o que leva ao desbalanceamento de carga entre os processadores, o que pode ser verificado com *Carnival*. Uma alteração do algoritmo de busca estava fora do escopo do nosso trabalho. A surpresa desta aplicação é que o desempenho obtido foi praticamente a metade do quasi-linear apresentado na literatura [LDCZ85], apesar de usarmos a mesma plataforma SP2 não utilizamos a versão de *TreadMarks* (1.0). A explicação dessa diferença de desempenho que ainda não confirmamos, é devida ao fato da nossa versão usar o protocolo UDP/IP ao invéz da biblioteca MPL que acessa diretamente a interface de rede. Se este for o caso, torna-se desejável para que ferramentas tais como *Carnival* verifique e informe ao programador medidas de calibragem de comunicação do sistema, por exemplo, tempo médio de troca de mensagens que permitiria avaliar o *overhead* do protocolo de comunicação utilizado.

D. IS

Nesta versão paralela de IS, cada processador tem um conjunto local de chaves que deverão ser ranqueadas globalmente. Na implementação utilizada, cada processador atualiza primeiramente um vetor local que armazena a densi-

dade das chaves locais. Em seguida cada processador atualiza um vetor compartilhado representando a densidade global das chaves. Ao final os processadores efetuam leituras neste vetor para posicionar suas chaves corretamente.

Os *speedups* iniciais se encontram na curva 1 da figura 2. *Carnival* aponta que uma barreira (bucksort_barrier2) é o escopo de maior importância dentro do tempo de execução da rotina de ordenação, com 50% do tempo de espera da aplicação. Também indicou o *lock bucksort_lockacquire*, como o escopo de maior importância (84%) para a causa de tempo de espera elevado nesta barreira. Mais ainda, o histograma deste *lock* indica que praticamente 100% do tempo de execucao desta operação é tempo de espera pelo *lock*.

Por outro lado, observando-se o código da aplicação, vemos que este *lock* protege o vetor compartilhado, e provoca o efeito de serialização que é refletido na barreira, ou seja, conforme vão terminando suas escritas, os processadores liberam o *lock* e esperam pelos outros na barreira.

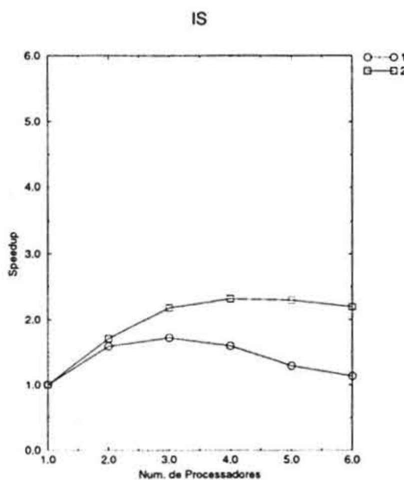

Fig. 2. Speedups obtidos com a depuração de desempenho de IS

Para diminuir este tempo de espera, dividimos o vetor compartilhado em *n* porções, correspondente aos *n* processadores. É realizado então um loop de *n* passos, onde cada processador atualiza uma porção diferente do vetor a cada iteração. Os *speedups* da aplicação para esta modificação melhorou sensivelmente como indicado pela curva 2 da figura 1.

Seguindo para a análise de tempo de espera, podemos perceber que o *loop* paralelo *writesharedkeyden* contribui com mais de 95% do tempo de espera total do programa. Este é o *loop* responsável pela atualização do vetor compartilhado. O *Charac Map* de *Carnival* indica que mais de 98% do tempo de espera neste *loop* se deve a espera por dados remotos decorrentes das falhas de acesso (faltas de página). Estas ocorrem pois os processadores devem conhecer as modificações feitas por seus vizinhos na iteração anterior.

Neste ponto, embora os *speedups* ainda sejam modestos não há mais nenhuma modificação imediata a ser feita.

IV. Conclusão

Neste artigo, avaliamos a eficácia da ferramenta *Carnival* para depurar desempenho de aplicações paralelas em ambiente *software* distributed shared memory (DSM), por programadores sem experiência prévia tanto em programação paralela como com *Carnival*. Nossos resultados baseados em duas aplicações do Splash2 (*Raytrace* e LU) e duas aplicações (IS e TSP) que acompanham *TreadMarks* mostram que *Carnival* é precisa no diagnóstico das causas e que fornece para o programador pistas para restruturação simples e eficientes do código que permitem aumentar o desempenho da aplicação. obtidos com as modificações nos trechos de código indicado por *Carnival*. Para as aplicações do SPLASH2, voltadas para *hardware* DSM, o aumento de desempenho alcançado foi expressivo e para as aplicações já otimizadas para *TreadMarks* o aumento foi modesto, como era de se esperar. Mostramos também que é importante que ferramentas de depuração de desempenho para *software* DSM apresentem medidas de calibragem do protocolo de comunicação utilizado pelo sistema *software* DSM pois podem restringir severamente o desempenho potencial das aplicações.

Dando continuidade ao trabalho, pretendemos utilizar *Carnival* em aplicações que permitam isolar e analisar melhor o efeito da comunicação (falhas de páginas) do que o de sincronização de forma a reestruturar a alocação dos dados para evitar o falso compartilhamento e a fragmentação, especialmente em outras classes de aplicações, tais como aplicações de servidores WEB e multimídia paralelos.

V. Agradecimentos

Gostariamos de agradecer a Lauro Whately pelo porte de LU para *TreadMarks*. Este trabalho foi realizado com o uso do IBM SP2 pertencente ao NCE/UFRJ.

Referências

[JSS97] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. *Proceedings of the sixth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 217–229, June 1997.

[KCDZ94] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Proceedings of the winter USENIX conference*, pages 115–132, January 1994.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *Proceedings*

of the Nineteenth International Symposium on Computer Architecture, pages 13–21, May 1992.

[Lar93] J. R. Larus. Efficient program tracing. *IEEE Computer*, pages 52–61, May 1993.

[LDCZ85] H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. *Proceedings of Supercomputing'95*, December 1985.

[MJ97] W. Meira Jr. *Understanding Parallel Program Performance using Cause-Effect Analysis*. PhD thesis, University of Rochester, 1997.

[MLHA97] W. Meira, T. J. LeBlanc, N. Hardavellas, and C. Amorim. Understanding the performance of dsm applications. *Proceedings of the Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC' 97)*, pages 198–211, February 1997.

[MLP96] W. Meira, T. J. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in carnival. *Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 1–10, May 1996.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[SGL94] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, pages 45–55, July 1994.