Implementação MPIC++ dos *kernels* NPB EP, IS e CG

Ricardo Leonarczyk[1], **Dalvan Griebler**[1,2]

[1] Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC)
Faculdade Três de Maio (SETREM), Três de Maio, Brasil.

[2]Escola Politécnica, Grupo de Modelagem de Aplicações Paralelas (GMAP),
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brasil.

ricardo.leonarczyk95@gmail.com,dalvan.griebler@acad.pucrs.br

Resumo. *Este trabalho busca contribuir com prévios esforços para disponibilizar os NAS Parallel benchmarks na linguagem C++, focando-se no aspecto memória distribuída com MPI. São apresentadas implementações do CG, EP e IS portadas da versão MPI original do NPB. Os experimentos realizados demonstram que a versão proposta dos benchmarks obteve um desempenho próximo da original.*

1. Introdução

Benchmarks desempenham um importante papel na avaliação e comparação de arquiteturas paralelas, pois proveem cargas de trabalho similares as de aplicações reais e com o benefício de serem amplamente estudados pela comunidade da computação de alto desempenho. Além do teste de arquiteturas, os *benchmarks* também tem sido usados para estudar características de diferentes bibliotecas paralelas, compiladores e ambientes de execução, o que pode requerer alterações no código ou sua reescrita em outra linguagem de programação. A disponibilidade de *benchmarks* em múltiplas linguagens não só tem impacto positivo para as comunidades que as utilizam, mas também é vantajoso para a comparação entre ferramentas de mesmo propósito provindas de diferentes linguagens.

A suíte de *benchmarks* provida pela NAS (*Nasa Advanced Supercomputing division*), conhecidos como *NAS Parallel benchmarks* (NPB), simula o comportamento de aplicações da área da dinâmica dos fluídos, sendo composta por um conjunto de *kernels* e pseudo aplicações [Bailey et al. 1991]. Os *kernels* focam em um conjunto de características, como o desempenho de comunicação ou de operações de ponto flutuante.

Desde a criação da suite NPB houve diversos esforços para portar seus *benchmarks* para outras linguagens de programação e/ou bibliotecas de paralelização diferentes das fornecidas pela NAS. No entanto, nenhum trabalho até agora (dentro do nosso conhecimento) portou o NPB com MPI na versão C++. Os trabalhos que produziram versões do NPB para arquiteturas distribuídas em C++ adotaram o modelo de programação PGAS (*Partitioned Global Address Space*). O trabalho de [Fürlinger et al. 2016] paralelizou o kernel DT com a biblioteca distribuída DASH. Os resultados mostram um aumento de até 24% em *speed-up* para o DASH DT quando comparado com a versão nativa do DT MPI, graças as operações unilaterais utilizadas em DASH. Já [Sakae and Matsuoka 2001], em um trabalho mais antigo, paralelizou os *kernels* IS e CG para uma extensão do C++ chamada de MPC++. O objetivo foi criar uma versão portável do MPC++ através do uso de MPI como camada de comunicação, substituindo a biblioteca de troca de mensagens especializada para redes Myrinet MP. No geral, Verificou-se que a implementação com

MPI pode substituir a que utiliza MP sem que as aplicações sofram perdas significativas em desempenho. Apesar de tanto a biblioteca DASH quanto a linguagem MPC++ serem capazes de utilizar MPI, estas o fazem indiretamente para trocas de mensagens requeridas por operações de alto nível do modelo PGAS. Esta abordagem difere-se da utilizada no presente trabalho, que faz uso direto da MPI para paralelizar os *benchmarks*.

Este trabalho faz parte de um esforço maior que tem como objetivo prover versões do NPB em C++ com bibliotecas para programação distribuída. É também uma extensão do trabalho de [Griebler et al. 2018], que apresenta uma implementação de 5 *kernels* NPB para a arquitetura multi-core. O objetivo neste artigo é apresentar a implementação dos *kernels* EP, IS e CG em C++ com MPI, e demonstrar seu desempenho em comparação aos *kernels* originais escritos em Fortran com MPI. Desta forma, na Seção 2 a implementação dos *kernels* é descrita, na Seção 3 os resultados são discutidos, e por fim, a Seção 4 apresenta as conclusões alcançadas.

2. Implementação

Para portar os *kernels* foram utilizadas como base as versões -sequenciais do NPB 3.3.1 em C++ disponibilizadas pelo trabalho de [Griebler et al. 2018]. Observando-se a estrutura do código Fortran dos NPB 3.4 paralelizados com MPI, fez-se as inserções das chamadas à MPI e as reestruturações necessárias para que o código ficasse semelhante ao máximo à versão Fortran MPI original. Levou-se em conta as diferenças inerentes entre as duas linguagens. O código resultante mantém-se em maior parte dentro do subconjunto C do C++, o que pode facilitar a portabilidade entre estas linguagens. Isso ocorreu de tal forma que o kernel IS pôde ser portado sem modificações significantes ao código na sua versão MPIC++.

As reestruturações citadas envolveram a implementação de refatorações feitas no NPB 3.4. Como por exemplo, a separação da definição de variáveis e algumas funções auxiliares em um arquivo de código fonte separado do principal (que contém a lógica do algoritmo) ou a opção de uso dos cronômetros de operações sendo definida através de variáveis de ambiente em vez de arquivos.

O kernel EP consiste na geração de um número determinado de pares de desvios gaussianos aleatórios que pode ser feita de forma independente, de modo que a comunicação só precisa acontecer na junção dos resultados ao final [Bailey et al. 1991]. Na implementação MPI, o número de iterações do laço principal responsável pela geração dos pares é dividido entre os processos, levando-se em conta que em caso de uma divisão com resto haverá alguns processos gerando mais pares que outros. Ao fim do laço, para cada processo, as variáveis sx e sy contém a soma dos desvios gaussianos gerados, e o vetor q contém a tabulação da quantidade de pares dentro do *square annulus*. É feita então a soma das variáveis e do vetor entre todos os processos para obter os valores finais, que são validados pelo processo raiz.

O kernel CG consiste na aplicação do método da potência inversa para a solução do sistema linear $Az = x$ no qual A é uma matriz esparsa com um padrão aleatório de números diferentes de zero [Bailey et al. 1991]. A matriz é particionada entre os processos, que precisam estar em número de base 2. Cada processo precisa calcular seus limites de linhas e colunas, gerar a submatriz esparsa em sua partição através da chamada à função $makea$ e inicializar o vetor x com valores iguais a 1. Após isso, o algoritmo entra no laço

Tabela 1. Resultados dos experimentos usando a classe C. São apresentados a média aritmética do tempo de 10 execuções e o desvio padrão.

	EP-CPP		EP		CG-CPP		CG		IS-CPP		IS	
Proc.	Média	σ	Média	σ	Média	σ	Média	σ	Média	σ	Média	σ
seq.	971.52	1.38	976.94	0.76	564.26	0.38	573.12	2.65	64.94	0.25	64.39	0.12
2	481.17	0.17	488.49	0.40	264.93	0.80	257.00	0.86	25.82	0.65	25.70	0.71
4	240.79	0.08	244.44	0.19	164.96	1.37	160.14	1.46	17.32	0.14	17.21	0.13
8	120.51	0.08	122.41	0.06	107.00	2.27	104.39	2.52	11.04	0.13	10.99	0.18
16	60.59	0.39	61.38	0.07	146.85	2.16	145.46	1.83	9.60	0.27	9.54	0.45
32	30.43	0.09	30.86	0.05	166.74	6.88	164.01	7.75	11.54	1.03	11.55	0.79

da potência inversa, onde ocorre a chamada à função `conj_grad`, responsável pela aplicação do método do gradiente conjugado. Nesta função, a MPI é utilizada para trocas de dados entre as partições nas reduções de vetores e escalares necessárias para multiplicar A por z, empregando comunicação irregular. Ao contrário do EP, no CG não são utilizadas as operações coletivas de MPI, apenas o `MPI_Send` e `MPI_Irecv` (recebimento assíncrono) para implementar as reduções entre grupos de processos. Posterior à aplicação do gradiente conjugado uma redução de soma é feita como parte da multiplicação dos vetores x e z para se obter o valor ζ, utilizado pelo processo raiz na validação da operação realizada pelo kernel.

3. Resultados

Os experimentos foram executados em um *cluster* de 4 nós, cada qual equipado com 2 processadores AMD Opteron(tm) de 6 núcleos físicos a 2.100GHz com suporte a *hyperthreading* desabilitado, 32GB de memória RAM e tendo o Ubuntu Server como sistema operacional. Cada nó do *cluster* possui agregação de 4 links (modo *bonding round-robin*) Gigabit Ethernet (largura de banda teórica é de 4Gb). O mapeamento dos processos MPI foi em *round robin*, no qual sempre um processo é atribuído a um nó diferente. Assim, sempre mais de um nó (e por consequência a rede) é utilizado. O número de processos manteve-se na potência de dois (restrição imposta pelo CG).

O *speed-up* teve como base a versão sequencial dos *benchmarks* originais e da versão C++ de [Griebler et al. 2018]. A versão sequencial para Fortran foi obtida através da compilação da versão OpenMP com o OpenMP desativado, como sugerido pela NAS para o NPB 3.4. Para o cálculo do *speed-up* foi considerada a média aritmética de um conjunto de 10 execuções para cada combinação de *benchmark*, classe e número de processos. A média de tempo reportada na Tabela 1, refere-se ao tempo total (execução e comunicação) do processo que mais demorou a terminar. Devido à limitações de espaço, o speed-up não é mostrado na tabela. É possível obtê-lo através dos valores das médias de tempo. Os *benchmarks* foram compilados com GFortran, GCC e G++, na versão 7.4 e com nível de otimização 3 (*flag* -O3). Por padrão os *benchmarks* do NPB 3.4 utilizam o gerador de números aleatórios `randi8`, escrito em Fortran. É incluso também o gerador `randdp`, que possui versões em C (para uso no IS) e Fortran. O `randi8` tem maior desempenho, porém ainda não foi portado para C++. Desta forma, os benchmakrs escritos em fortran foram configurados para usar o `randdp`. Utilizou-se nos experimentos as classes A, B e C, porém no artigo apenas é mostrado o resultado para a classe C, por esta possuir uma carga de trabalho maior.

Para o *kernel* EP, a versão C++ apresentou um aumento no *speed-up* próximo de 1% comparado a versão Fortran. Observando a medição de tempo apenas da geração de

números aleatórios, foi possível constatar que a função responsável por gerar estes números, originada do C, obteve uma leve vantagem em tempo de execução sobre a mesma escrita em Fortran. No entanto, a maior diferença de tempo esteve na geração dos pares gaussianos.

Já para o CG, a versão Fortran alcançou *speed-ups* entre 2,46% a 4,42% maiores do que a versão C++, dependendo do número de processos. Percebe-se que o CG C++ possui escalabilidade similar a da versão Fortran, obtendo maior desempenho na execução com 8 processos. A variabilidade nas medições de tempo foi maior para o CG, que obteve desvios padrão até 7,75. Mas ainda que presentes, as variações seguem o mesmo padrão para as duas versões, aumentando ou diminuindo de forma semelhante de acordo com o número de processos. Um motivo para a presença de tais variações pode ser o grande número de comunicações irregulares sendo feitas na função do gradiente conjugado, que em virtude da configuração de distribuição de processos escolhido, faz uso constante da rede. O IS obteve um desempenho semelhante nas duas versões (C e C++), como esperado. O IS C++ apresentou um tempo de execução ligeiramente maior, todavia alcançou um aumento similar no *speed-up*.

4. Conclusões

Este trabalho apresentou a implementação e os resultados da versão MPI dos *kernels* NPB CG, EP e IS para C++, e comparou seu desempenho ao dos *kernels* originais. Verificou-se que os *kernels* C++ obtiveram desempenho similar às suas versões originais em C e Fortran, embora a semelhança seja maior em escalabilidade do que em desempenho. O EP C++ apresentou uma melhora de cerca de 1% no *speed-up*, enquanto o CG C++ sofreu a maior penalidade no desempenho, sendo o *speed-up* da implementação original 4,42% maior em alguns casos. O desempenho do IS foi o que mais se aproximou do original, como já era esperado. Investigações a serem feitas podem revelar o motivo para a diferença em desempenho das versões do CG. Em trabalhos futuros pretende-se portar o restante dos *kernels* paralelizados com MPI, e explorar o uso de outras bibliotecas para a programação distribuída em C++.

Referências

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

Fürlinger, K., Fuchs, T., and Kowalewski, R. (2016). Dash: A c++ pgas library for distributed data structures and parallel algorithms. In *IEEE 18th Intern. Conf. on High Performance Computing and Communications*, pages 983–990. IEEE.

Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient NAS benchmark kernels with C++ parallel programming. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 733–740. IEEE.

Sakae, Y. and Matsuoka, S. (2001). MPC++ Performance for Commodity Clustering. In *International Conference on High-Performance Computing and Networking*, pages 503–512. Springer.