Laboratório de Projeto de Sistemas Computacionais

Grupo 1 *
Cristina Freitas Bazzano RA 135368
Flávio Altinier Maximiano da Silva RA 135749
Gabriel Militão Vinhas Lopes, RA 135801
Marcos Massayuki Kobuchi RA 136823

26 de Junho de 2015

1 Introdução

Um processador multicore é um único chip contendo dois ou mais núcleos de processamento, permitindo que diferentes tarefas possam ser divididas, criando assim um ambiente multitarefa.

Processadores multicore são importantes no processamento paralelo, pois permitem que um mesmo software possa ser executado simultaneamente em diferentes núcleos, de forma a melhorar o desempenho no processamento. No entanto, o software deve ser desenvolvido levando em consideração a capacidade e as limitações do multiprocessamento.

Neste projeto o objetivo é modelar um sistema com 8 núcleos utilizando a biblioteca SystemC, simulando na arquitetura ArchC. Para a análise dos resultados gerados pelo ambiente multitarefa, desenvolvemos e executamos um software que busca tirar proveito desse paralelismo.

2 Metodologia

Nos foi fornecido uma plataforma base que possui um processador MIPS monocore capaz de executar softwares compilados em archc. Para a modelagem do processador de 8 núcleos, foi necessário a modificação de diversos arquivos que compõem o sistema.

Todas as modificações foram pensadas para a execução do software desenvolvido por nós *somaVetor.c* para tirar proveito desse paralelismo.

*Repositorio Git http://bit.ly/1BsFWGY

2.1 Software MapReduce em Paralelo

O MapReduce é um modelo de programação desenvolvido pela Google projetado para processar grandes volumes de dados em paralelo, dividindo o trabalho em um conjunto de tarefas independentes.

Dadas estas características, implementamos uma versão simplificada desse modelo, a fim de analisar seu desempenho no nosso sistema multicore. O problema escolhido é a soma de dois vetores de inteiros, no arquivo *somaVetor.c*. Os vetores possuem 256000 posições, inicializadas com valor 1. O objetivo do programa é encontrar a soma de todos os elementos dos dois vetores.

Primeiramente, o **processador 0** inicializa os vetores 1 e 2 que serão somados, e em seguida "acorda"os outros cores. Então começa a fase Map do problema: cada processador é encarregado de somar um oitavo dos vetores 1 e 2 em um outro vetor de 256000 posições resultMap. Em seguida, cada processador usa uma variável global para avisar os outros de que terminou.

Em seguida, começa a fase Reduce: cada processador é encarregado de somar todos os elementos em um oitavo do vetor ResultMap, em um vetor de oito posições - cada uma guarda a soma total dos elementos de cada oitava de ResultMap - chamado de ResultReduce.

Cada processador que termina sua parte incrementa um contador de processadores "finalizados". Quando todos terminarem, o processador 0 é encarregado de varrer esse vetor ResultReduce de apenas 8 posições e somar seus elementos; esse processador imprime então na tela o resultado da soma total.

Para fins de comparação, desenvolvemos também outro arquivo de testes, somaVetorSerial.c. Nesse programa apenas o processador 0 é encarregado de fazer todas as somas. Dessa forma, podemos comparar se para os fins dessa aplicação o paralelismo simulado é vantajoso, levando em consideração o tempo de execução da aplicação.

2.2 Inicialização do sistema multicore

Na inicialização de um processador multicore primeiramente apenas um dos cores é inicializado e este é responsável por inicializar todos os demais.

A instanciação da simulação dos 8 processadores utilizados no nosso sistema é feita no *main.cpp*, onde são criados 8 simuladores MIPS numerados de 0 a 7. Esses processadores comunicam-se com a memória através do envio de *requests* para um *bus* comum, ao qual eles são ligados após sua instanciação.

O passo seguinte é inicializar apenas o **processador 0**, dando *pause* nos demais processadores. Antes de iniciar a simulação passamos para o *bus* o endereço de cada processador para permitir o envio de mensagens a eles.

Feito isso, a simulação é iniciada e nela o processador 0, após fazer as inicializações do problema, inicializa os demais processadores para começar o cálculo da solução do problema em paralelo (como descrito anteriormente).

2.3 Comunicação via barramento

Todo o acesso a memória da nossa simulação passa pelo barramento. Utilizamos esse fato para realizar a comunicação entre os processadores. Definimos para tanto que os endereços de memória da forma 0x7000xx serão destinados a pausar os processadores xx/4 e os endereços de memória da forma x7010xx serão destinados a continuar a execução dos processadores xx/4.

Dessa forma um processador pode pausar ou continuar a execução de outro processador ao tentar alterar um endereço de memória.

2.4 Controle de concorrência

Com a presença de diversos processadores operando sobre mesmos módulos, como a memória e o barramento, podem ocorrer problemas de consistência visto que eles estão competindo pelo uso dos mesmos recursos. Um processador pode, por exemplo, ler uma região da memória, outro processador altera essa região, e o primeiro processador agora está operando sobre um valor desatualizado.

Para contornar esse problema, podemos implementar algumas soluções em hardware e em software.

2.4.1 LL/SC

Para implementar em hardware, alteramos o arquivo mips_isa.cpp adicionando a ele dois novos comportamentos (ac_behaviour) chamados *Load Link (ll)* e *Store Condition (sc)*. Tais instruções são utilizadas em multithreading para realizar sincronização.

A instrução load link realiza uma leitura, retornando o valor corrente que está sendo armazenado em uma certa posição na memória. Já uma subsequente chamada de *store conditional* na mesma posição de memória armazenará um novo valor apenas se não tiver sido realizada nenhuma atualização nesse valor desde a chamada de *load link* e armazena o resultado (se houve escrita ou não) no *rt*. Assim, juntas, elas implementam uma operação de leitura e escrita atômica sem o uso de *lock*.

Além disso, para complementar a implementação das instruções, a definição de tais comportamentos foi adicionada ao arquivo mips_isa.ac, com a observação de que o código de operação (opcode) da instrução *Load Link* é 0x30 e o código de operação da instrução *Store Condicional* é 0x38.

Para que os softwares desenvolvidos sejam compilados para aceitar esse novo comportamento, ao se compilar os códigos fonte, adicionamos a flag -march=mips32 para que seja usado a versão 2.0 do mips, no qual existe essas instruções.

2.4.2 Lock

Para a implementação do *lock*, utilizamos as instruções *Load Link e Store Conditional* da seguinte forma:

do {

```
            one = 1;
            ll r, (*plock)
            sc (*plock), one
        } while (!(r == 0 && one == 1  && *plock == 1));
```

Assim, o processador continua esperando o *lock* enquanto não consegue:

- Ler o valor zero do *lock*;
- Escrever o valor um no *lock* sem ninguém ter modificado seu valor

Desta forma, ele sai da operação de *lock* se nenhum processador tinha o *lock* e ele conseguiu escrever no *lock*, ou seja, o processador obtém o *lock*. Além das operações de *lock* e *unlock*, a operação *lock_init* (que inicializa o *lock*) foi implementada e é chamada somente pelo processador 0.

2.5 Software Offloading

Primeiramente, para implementar um software offloading genérico é necessário criar uma classe que representa tal estrutura. Em *main.cpp* o *software offloading* é instanciado e conectado ao *bus*. Já em *bus.cpp* cria-se um endereço físico associado a este módulo de forma a possibilitar aos outros componentes uma forma de se comunicar com o *soft. offloading*.

Desta forma, com o objetivo de otimizar instruções de soma de vetores grandes, busca-se implementar um módulo em hardware dedicado a realizar este tipo de operação. Para tanto, basta que um processador que tenha interesse em utilizar o software *offloading* obtenha o *lock* deste módulo, passe os vetores como parâmetro e aguarde a resposta no barramento.

3 Resultados

Como caracterizado na metodologia, nesta seção devemos comparar a execução da soma de todos os elementos de dois vetores para o MIPS com 8 *cores* com relação ao de apenas um *core*.

O resultado observado foi bastante instigante. Para o problema resolvido de forma paralela, utilizando os oito *cores*, após a realização de 10 testes obtivemos um tempo médio de execução de 18.11±0.04s; para o problema serial, os resultados observados foram de 12.06±0.04s - substancialmente mais rápidos.

Atribuímos esse melhor desempenho devido ao maior *overhead* do problema paralelo; além disso, acreditamos que embora a aplicação desenvolvida simule 8 processadores, a simulação seja realmente feita em apenas um *core* da máquina real do laboratório em que foi testada: dessa forma, o processador real passa boa parte do tempo controlando as interações entre os processadores simulados, causando um maior atraso.