Um Mecanismo para Programação Multithreaded em Arquitetura Multicore

Benedito Barbosa Ribeiro Neto[1], Adalton de Sena Almeida[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia (IFPI) – Teresina – PI – Brasil

[2]Universidade Federal de Pernambuco (UFPE) – Recife – PE – Brasil

bbrneto@gmail.com, adaltonsena@gmail.com

Abstract. *Increased energy loss, limit the extraction of parallelism in instruction level and complexity of the project led to the development of multicore microprocessors. To improve existing applications and / or provide the implementation of new applications based on a parallel programming model, this paper aims to propose a mechanism for multithreaded programming that is tangible to most developers, and providing the best possible performance to running applications that use processing-intensive microprocessor architecture of multicore.*

Resumo. *Aumento da energia dissipada, limite na extração de paralelismo no nível de instruções e complexidade de projeto levaram ao desenvolvimento de microprocessadores multicore. Para melhorar as aplicações existentes e/ou proporcionar a implementação de novas aplicações baseadas em um modelo paralelo de programação, este trabalho tem por objetivo propor um mecanismo para programação multithreaded que seja tangível para a maioria dos programadores, e que proporcione o melhor desempenho possível ao executar aplicações que utilizam processamento de forma intensiva em microprocessadores de arquitetura multicore.*

1. Introdução

Durante as últimas décadas, o desempenho alcançado pelos microprocessadores cresceu exponencialmente. Esse crescimento deveu-se principalmente ao avanço da tecnologia de fabricação que possibilitou a construção de transistores menores e mais rápidos. No entanto, mesmo seguindo a Lei de Moore[1], fatores novos estão limitando o crescimento do desempenho dos microprocessadores: aumento da energia dissipada, limite na extração de paralelismo no nível de instruções e complexidade de projeto. A tendência atual é desenvolver projetos mais simples, com freqüência de operação mais baixa, e integrar em um mesmo chip dois ou mais núcleos de processamento (RIGO et al, 2007).

Em arquiteturas de um só núcleo (*singlecore*), o aumento da freqüência de operação dos microprocessadores implicava um aumento proporcional no desempenho do *software* existente. Em arquiteturas de múltiplos núcleos (*multicore*), isso deixa de ser verdade. Para fazer uso do potencial oferecido pelo *hardware* atual, o *software* deve ser paralelizado. Dessa forma, o modelo seqüencial de programação deve ser alterado por um modelo paralelo (RIGO et al, 2007).

[1] Em 1965, Gordon Moore especulou que o número de transistores utilizados em cicuitos integrados iria dobrar anualmente. Em 1975, ele alterou sua projeção, estipulando que esse número dobraria em intervalos de dois anos. Essa previsão, amplamente conhecida como Lei de Moore, tem-se mantido até os dias atuais e se tornou a força motora que alavancou o avanço da indústria de semicondutores (RIGO et al, 2007).

O modelo paralelo de programação é tão antigo quanto o modelo seqüencial, mas é inerentemente mais difícil, principalmente quanto ao desafio de se produzir um código correto, robusto e seguro. O modelo mais utilizado atualmente é o *multithreaded*, no qual um processo é uma abstração para um programa em execução. Todo processo consiste de um espaço de endereçamento próprio, recursos do sistema, contexto de *hardware* e outras informações mantidas pelo sistema operacional (RIGO et al, 2007).

Thread é um segmento de execução independente dentro de um processo. O estado global (como espaço de endereçamento e recursos do sistema) é compartilhado por todas as *threads* criadas a partir de um mesmo processo. Cada *thread* também possui um contexto privado, com sua pilha, registradores e outras informações (RIGO et al, 2007). A Figura 01 ilustra uma aplicação composta por várias *threads*.

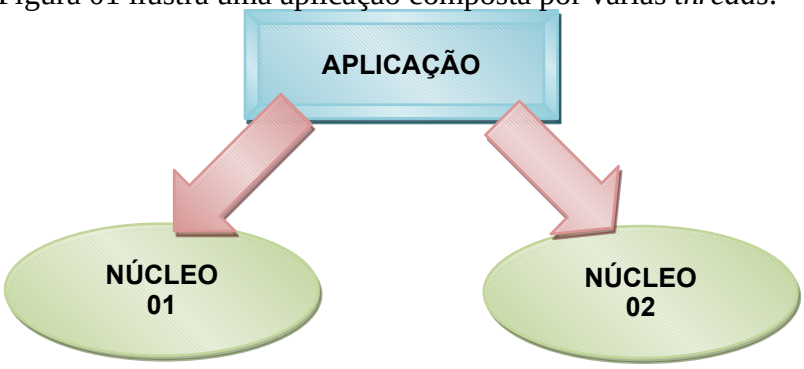


Figura 01 – Aplicação composta por várias *threads* sendo executada em um microprocessador *multicore* (Adaptado de CARDOSO et al, 2005).

O sucesso do modelo *multithreaded* decorre da associação quase direta entre as abstrações oferecidas pelo modelo e a forma como o *hardware* atual as implementa. Atualmente, suporte para programação *multithreaded* pode ser encontrado nativamente em linguagens, bibliotecas, compiladores, sistemas de execução ou uma combinação desses (RIGO et al, 2007).

Aplicações escritas de forma que não exponha explicitamente o paralelismo existente podem até experimentar redução no desempenho, já que na arquitetura *multicore* os microprocessadores têm projetos mais simples e freqüência de operação mais baixa (RIGO et al, 2007). Este trabalho tem por objetivo propor um mecanismo para programação *multithreaded* que seja tangível para a maioria dos programadores, e que proporcione o melhor desempenho possível ao executar aplicações que utilizam processamento de forma intensiva em microprocessadores de arquitetura *multicore*.

2. Materiais e Métodos

Segundo Rigo et al (2007), é comum paralelizar iterações de um laço, pois cada iteração independente pode ser executada por uma *thread*. Laço é uma estrutura de repetição presente em linguagens de programação, composta por uma condição e um bloco de código. Verifica-se a condição, e caso seja verdadeira, o bloco de código é executado. Após o final da execução, a condição é verificada novamente, e caso ela ainda seja verdadeira, o bloco de código é executado novamente. Essa estrutura pode ser pré-testada, pós-testada, com variável de controle ou usada para iterar itens de uma coleção.

Desenvolvido em JAVA, a geração de lotes do IAPEP Saúde é um fluxo seqüencial formado pela iteração de itens de uma coleção. O IAPEP Saúde é o plano de assistência familiar oferecido pelo Instituto de Assistência e Previdência do Estado do

Piauí (IAPEP) aos servidores públicos estaduais e seus dependentes. Cada segurado tem acesso à atendimentos médicos e odontológicos através dos prestadores credenciados (hospitais, clínicas, consultórios e ambulatórios). Cada atendimento, representado por uma guia médica ou odontológica, armazena a data de atendimento do segurado e o valor total dos procedimentos médicos realizados por ele. Uma guia sempre está associada a um único prestador, pois ela é o instrumento utilizado para auditoria e pagamento do mesmo.

O IAPEP Saúde trabalha em regime de competência, isto é, o prazo estabelecido pelo plano para o registro de guias. Em outras palavras, competência é o período entre os dias 26 do mês anterior e 25 do mês atual. Por exemplo, a competência do mês de Abril compreende o período entre os dias 26 de Março e 25 de Abril. No final de uma competência, é necessário gerar um lote para cada prestador. O lote gerado facilita o trabalho dos auditores do plano, pois reúne as guias registradas por um prestador em uma determinada competência. A Figura 02 apresenta as classes envolvidas.

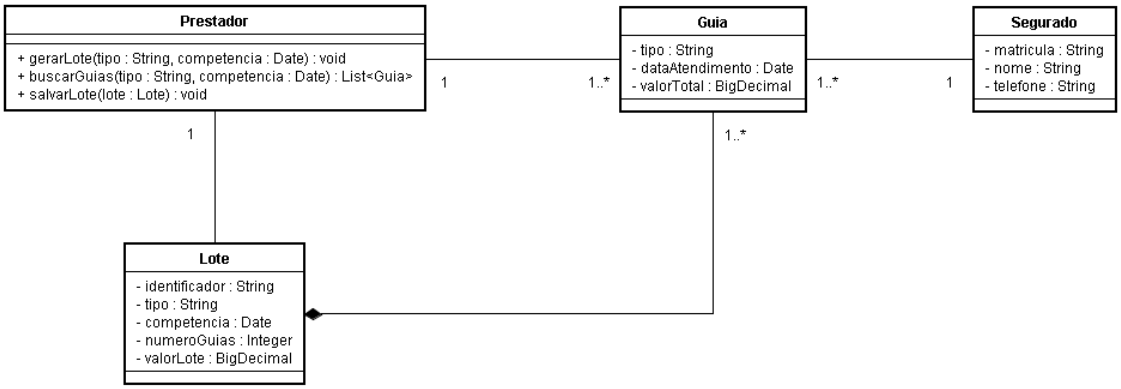


Figura 02 – Diagrama das classes envolvidas na geração de lotes.

Na Figura 02, o método buscarGuias() definido na classe Prestador retorna todas as guias registradas por um prestador em uma competência. Essas guias são associadas a um lote por meio do método gerarLote(). O lote possui um identificador para garantir a sua unicidade. Além disso, ele armazena a quantidade de guias associadas, através do atributo numeroGuias, e o somatório do valor total de cada uma, através do atributo valorLote. Para finalizar, o lote é salvo no banco de dados pelo método salvarLote(). A Figura 03 apresenta um exemplo da geração de lotes do IAPEP Saúde.

```
public class FluxoSequencial {

    public static void main(String[] args) throws Exception {
        String tipo = Atendimento.MEDICO.getDescricao();
        Date competencia = Utils.gerarCompetencia("01/2008");

        Queue<Prestador> prestadores = FluxoSequencial.buscarPrestadores(tipo, competencia);

        for (Prestador prestador : prestadores) {
            prestador.gerarLote(tipo, competencia);
        }
    }
```

Figura 03 – Fluxo seqüencial para geração de lotes.

Na Figura 03, o método buscarPrestadores() acessa o banco de dados e retorna os prestadores (entidades) que registraram atendimento médico entre os dias 26/12/2007

e 25/01/2008. Em seguida, cada prestador terá o método gerarLote() executado por meio de um laço. Nesse caso em específico, o laço chama-se *foreach*.

A fim de paralelizar as iterações desse laço, identificou-se o Padrão *Observer*. Segundo Booch et al (2005), todas as aplicações bem-estruturadas são repletas de padrões. Padrão é uma solução comum para um problema básico em um determinado contexto. Padrões de projeto são partes importantes do vocabulário dos programadores, pois fornecem uma linguagem compartilhada que maximiza a comunicação, o aprendizado e a troca de experiências.

O Padrão *Observer* mantém os objetos atualizados quando algo importante ocorre. Um serviço de assinatura de jornais é uma boa maneira de visualizá-lo. Nesse cenário, uma editora (sujeito) entrega novas edições de um jornal para seus assinantes (observadores). Enquanto forem assinantes, continuarão recebendo jornais. Ao cancelar a assinatura, o assinante deixa de receber os jornais, pois deixarão de ser entregues pela editora (FREEMAN e FREEMAN, 2007).

Tipicamente, os padrões de projeto são representados por colaborações. As colaborações têm dois aspectos: uma parte estrutural que especifica uma sociedade de classes, interfaces e outros elementos que trabalham em conjunto para fornecer algum comportamento cooperativo maior do que a soma de todas as suas partes; e a parte comportamental que especifica a dinâmica de como esses elementos interagem (BOOCH et al, 2005).

A parte estrutural do mecanismo proposto por este trabalho é apresentada pela Figura 04. Trata-se de um diagrama de classes modelado a partir do Padrão *Observer*, no qual a classe *Master* representa o observador e a classe *Slave* o sujeito. A parte comportamental é apresentada pela Figura 05. Trata-se de um diagrama de seqüência, cujas mensagens estão relacionadas a métodos visíveis na parte estrutural.

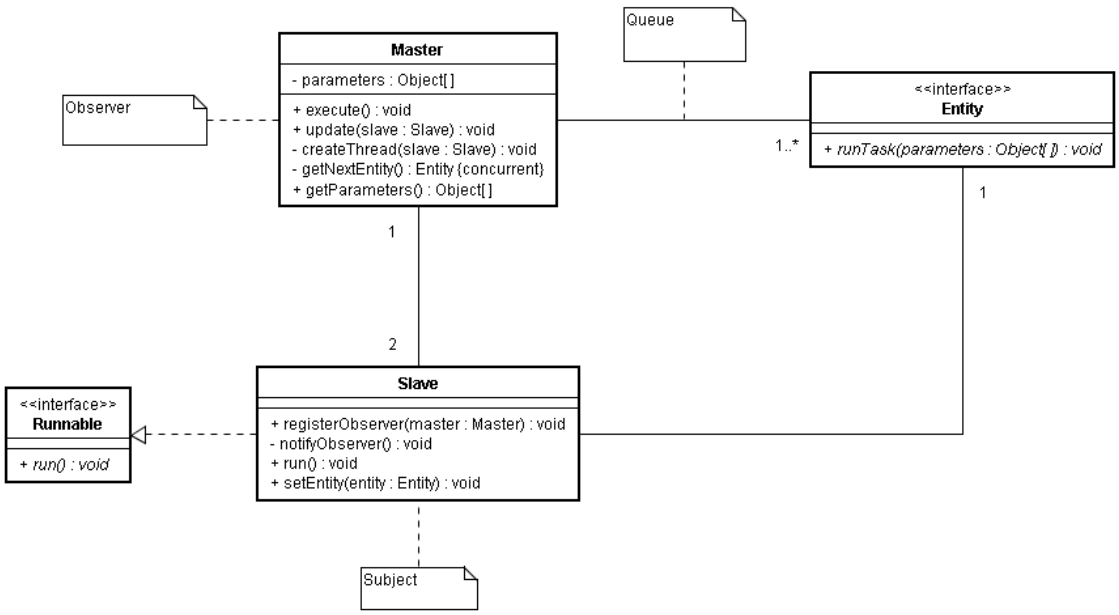


Figura 04 – Diagrama de classes modelado a partir do Padrão *Observer*.

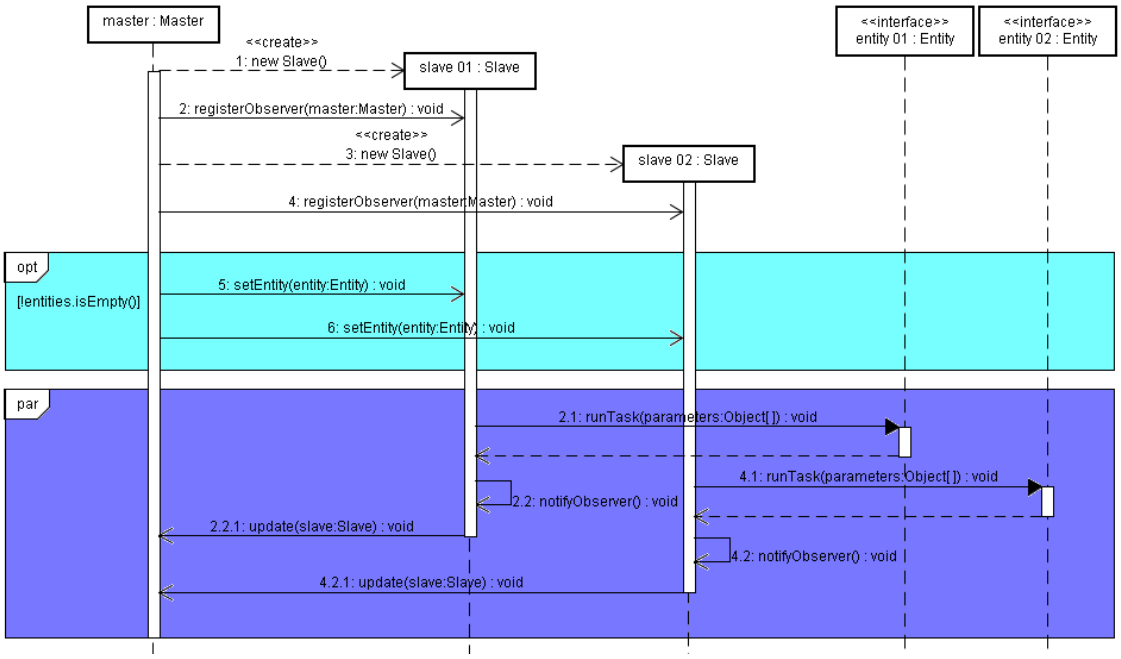


Figura 05 – Diagrama de seqüência do mecanismo proposto por este trabalho.

De acordo com os diagramas, um objeto da classe *Master* (mestre) escalona uma coleção de entidades (*Entity*) entre dois objetos da classe *Slave* (escravo). Essas entidades estão organizadas em fila (*Queue*) e compartilham a mesma tarefa, porém sobre diferentes partes de um conjunto de dados. Essa tarefa é definida pelo método *runTask()*, cujos parâmetros são atribuídos no construtor da classe *Master* e retornados pelo método *getParameters()*. Cada entidade terá sua tarefa executada por um escravo em uma *thread* diferente, através do método *run()*, definido pela interface *Runnable*.

Ao invocar o método *execute()*, o mestre instancia dois escravos e estabelece uma relação um-para-muitos através do método *registerObserver()*, conforme ao Padrão *Observer*. Em seguida, o método *createThread()* atribui uma entidade ao escravo por meio do método *setEntity()*, além de criar e iniciar uma nova *thread*, que executará o método *runTask()* da entidade. Após concluir a execução, o escravo utiliza o método *notifyObserver()* para informar ao mestre que a tarefa acaba de ser concluída. O método *update()* indica se ainda há uma entidade na fila. Se não houver, o processamento é concluído. Caso contrário, o método *getNextEntity()* encaminha ao escravo a próxima entidade. A Figura 06 apresenta a geração de lotes utilizando o mecanismo proposto.

```
public class FluxoParalelo {

    public static void main(String[] args) throws Exception {
        String tipo = Atendimento.MEDICO.getDescricao();
        Date competencia = Utils.gerarCompetencia("01/2008");

        Queue<Entity> prestadores = FluxoParalelo.buscarPrestadores(tipo, competencia);

        Master master = new Master(prestadores, tipo, competencia);
        master.execute();

    }
```

Figura 06 – Fluxo paralelo para geração de lotes.

3. Resultados e Discussão

No modelo *multithreaded*, *threads* comunicam-se através da memória compartilhada. Acessos às mesmas posições de memória precisam ser explicitamente sincronizados para evitar interferências entre *threads*. As formas de sincronização atuais empregam bloqueios, que do ponto de vista dos processadores *multicore*, limitam o paralelismo (RIGO et al, 2007). Para aumentar o nível de paralelismo, o mecanismo descrito adotou um único método sincronizado: *getNextEntity()*. Isso impede que os dois escravos executem a tarefa da mesma entidade. Dessa forma, menos bloqueios são utilizados, provendo estabilidade e diminuindo o risco de *deadlock*. Assim, o programador não precisa garantir a sincronização, pois o controle de acesso à memória compartilhada é feito automaticamente pelo mestre.

Na arquitetura *singlecore*, o processador executa somente uma tarefa por vez. Sabendo disso, entra em cena o escalonador de *threads*, que reúne as *threads* a serem executadas e faz o processador alternar a execução de cada uma delas. Na arquitetura *multicore*, a diferença é que o escalonador tem dois ou mais núcleos de processamento para executar suas *threads*, proporcionando execuções verdadeiramente paralelas. Segundo Sierra e Bates (2007), não é possível controlar o tempo de execução de uma *thread* e qual será a próxima *thread* a ser executada. Essas escolhas são feitas pelo escalonador da Máquina Virtual JAVA (JVM) e, mais amplamente, pelo escalonador do Sistema Operacional (SO). Primeiro, a própria JVM tem que ser o processo sendo executado pelo SO. Somente depois, a execução das *threads* será alternada pelo processador. É importante lembrar que as implementações do escalonador são diferentes para JVMs distintas e até executar a mesma aplicação no mesmo computador pode produzir resultados diferentes.

Para avaliar o desempenho da geração de lotes do IAPEP Saúde e o uso dos recursos de *hardware* disponíveis, utilizou-se um computador com 1.0 GB de memória, processador Intel de dois núcleos (2.0 GHz de freqüência cada núcleo) e sistema operacional Fedora 8. O espaço amostral para execução dos fluxos é formado por guias registradas em 2008. As figuras 07 e 08 mostram o desempenho obtido pela geração dos lotes de atendimento médico e odontológico, respectivamente. A análise dos dados se resume na comparação entre o tempo gasto pelo fluxo seqüencial e o tempo gasto pelo fluxo paralelo para geração dos lotes de cada competência.

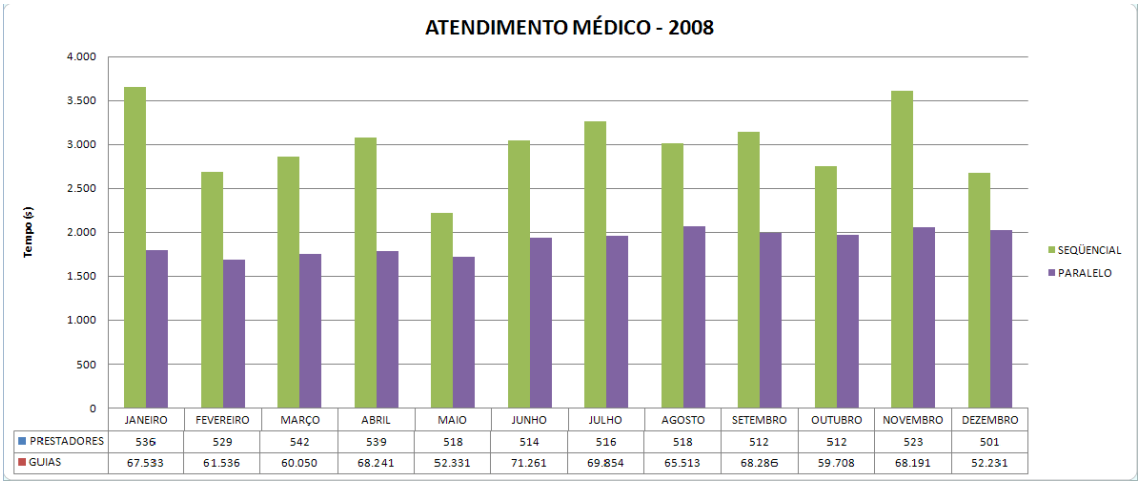
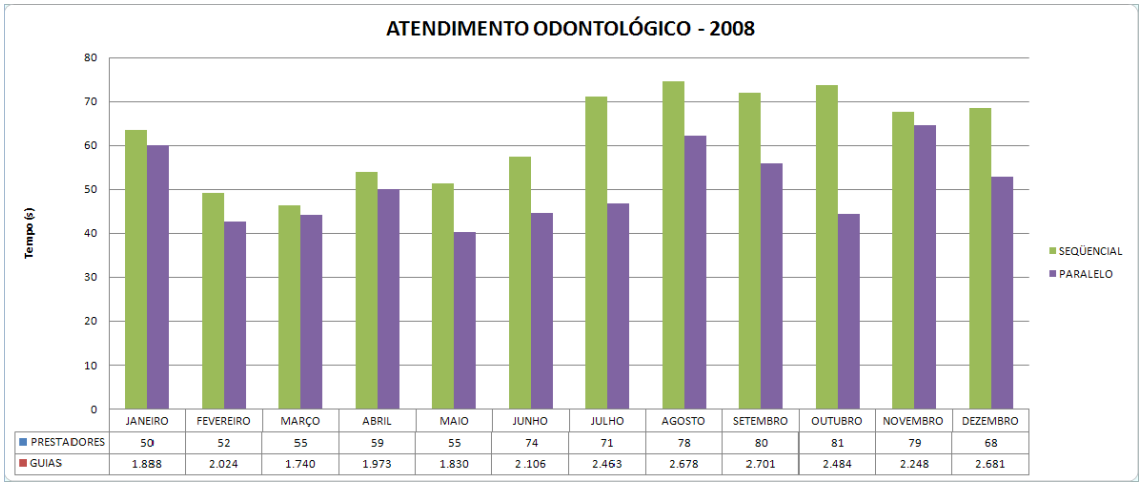


Figura 07 – Geração dos lotes de atendimento médico em 2008.



	JANEIRO	FEVEREIRO	MARÇO	ABRIL	MAIO	JUNHO	JULHO	AGOSTO	SETEMBRO	OUTUBRO	NOVEMBRO	DEZEMBRO
PRESTADORES	50	52	55	59	55	74	71	78	80	81	79	68
GUIAS	1.888	2.024	1.740	1.973	1.830	2.106	2.463	2.678	2.701	2.484	2.248	2.681

Figura 08 – Geração dos lotes de atendimento odontológico em 2008.

Observa-se, por exemplo, que 536 prestadores registraram 67.533 guias médicas em janeiro de 2008. A geração dos lotes dessa competência foi concluída após 3.655 segundos pelo fluxo seqüencial, enquanto o fluxo paralelo concluiu em 1.794 segundos. A diferença entre esses tempos (3.655s - 1.794s = 1861s) é equivalente à 50,92% do tempo gasto pelo fluxo seqüencial. O somatório dos percentuais de cada competência dividido pelo total de competências é igual a média total. Para o atendimento médico, a média total é de 36%. Para o atendimento odontológico, a média total é de 17,9%.

A Figura 09 mostra o uso dos núcleos de processamento durante a execução do fluxo seqüencial. De maneira alternada, cada núcleo utiliza até mais de 80% de sua capacidade de processamento. A Figura 10 mostra o uso dos núcleos de processamento durante a execução do fluxo paralelo. Nesse caso, cada núcleo utiliza menos de 40% de sua capacidade de processamento. Isso permite que outras aplicações possam ser executadas simultaneamente.

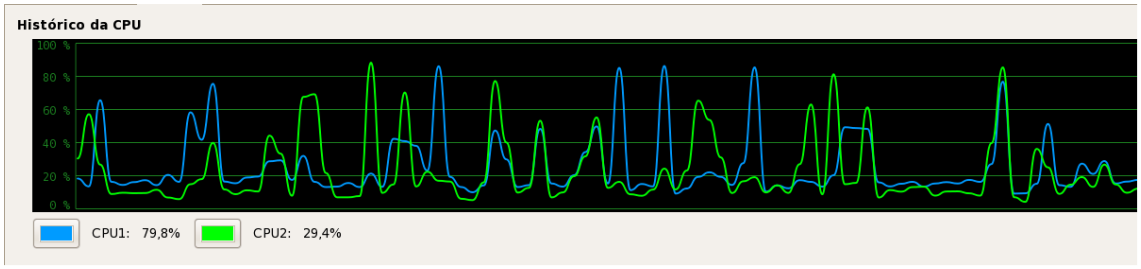


Figura 09 – Uso dos núcleos durante a execução do fluxo seqüencial.

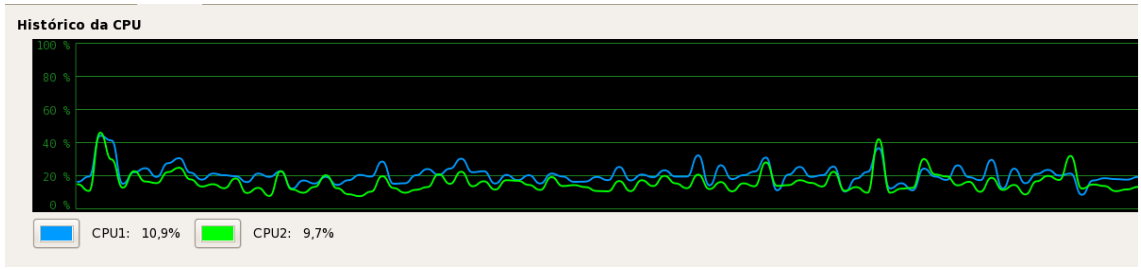


Figura 10 – Uso dos núcleos durante a execução do fluxo paralelo.

4. Conclusão

Este trabalho propôs um mecanismo para programação *multithreaded* modelado a partir do Padrão *Observer*. Uma vez que bloqueios restringem o nível de paralelismo em microprocessadores *multicore*, esse mecanismo mostra-se tangível para a maioria dos programadores, por aumentar o nível de paralelismo e controlar o acesso à memória.

Inicialmente, modelou-se elementos que permitem aos programadores visualizar, especificar, construir e documentar os artefatos de uma aplicação *multithreaded*. Esses elementos evitam, através de um único método sincronizado, interferências entre *threads* que compartilham um mesmo objeto. Dessa forma, menos bloqueios são usados, diminuindo o risco de erros e produzindo um código correto, robusto e seguro.

É importante ressaltar que o presente trabalho adotou a seguinte solução: substituir um bloqueio por uma abstração de nível mais alto. Essa solução não implica a eliminação total de bloqueios. Isso significa que o mecanismo exibe uma interface livre de bloqueios para o programador, embora sua implementação ainda os use.

Finalmente, para avaliar o desempenho desse mecanismo e o uso dos recursos de *hardware* disponíveis, utilizou-se a geração de lotes do IAPEP Saúde como contexto específico de implementação. Em resumo, a avaliação de desempenho mostra que o fluxo paralelo obteve melhor desempenho que o fluxo seqüencial, em todas as competências de 2008 tanto no atendimento médico quanto no odontológico.

Em média, a diferença entre o tempo gasto pelo fluxo seqüencial e o tempo gasto pelo fluxo paralelo para geração dos lotes é de 36% para o atendimento médico e de 17,9% para o atendimento odontológico. Esses valores revelam que o mecanismo proposto tem melhor desempenho em aplicações que utilizam processamento de forma intensiva, pois quanto maior o número de entidades, maior o número de *threads*.

Mesmo não sendo possível controlar o tempo de execução de uma *thread* e qual será a próxima *thread* a ser executada pelo microprocessador, verificou-se que outras aplicações terão melhor desempenho ao serem executadas simultaneamente à geração dos lotes, pois cada núcleo utiliza menos de 40% de sua capacidade de processamento.

REFERÊNCIAS BIBLIOGRÁFICAS

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML:** Guia do Usuário. 2. ed. Rio de Janeiro: Elsevier, 2005.

CARDOSO, Bruno. ROSA, Sávio. FERNANDES, Tiago. **Multicore**. 2005. Disponível em: http://www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2005/Trabalho/g07-multicore.pdf. Acesso em: 20 abr. 2009.

FREEMAN, Eric. FREEMAN, Elisabeth. **Use a Cabeça! Padrões de Projeto**. 2. ed. Rio de Janeiro: Alta Books, 2007.

RIGO, Sandro; CENTODUCATTE, Paulo Cesar; BALDASSIN, Alexandro. **Memórias Transacionais:** Uma Nova Alternativa para a Programação Concorrente. 2007. Disponível em: http://www.sbc.org.br/sbac/2007/cdrom/papers/wscad/minicursos//33214_1.pdf. Acesso em: 16 mar. 2009.

SIERRA, Kathy. BATES, Bert. **Use a Cabeça! Java**. 2. ed. Rio de Janeiro: Alta Books, 2007.