Utilização de técnicas de paralelismo para desenvolvimento de uma ferramenta com alto desempenho para varreduras de dispositivos de rede, escrita em linguagem C utilizando as bibliotecas *Socket* e *OpenMP*

Cristian Cleder Machado
URI-FW
cristian@cristian.com.br

***Resumo* — O presente artigo tem como objetivo a implementação e análise de execução de um um *Port Scan* escrito em linguagem C de forma sequencial e outro de forma paralelizada utilizando como comunicação para varredura das portas de destino a biblioteca *Socket*. Os ambientes de simulação variaram entre Arquiteturas de Computadores, Estrutura da Rede e Número de *Threads* executadas em cada teste.**

I. INTRODUÇÃO

Devido ao rápido crescimento na utilização de rede de computadores e o surgimento de várias aplicações voltadas à mesma, percebe-se que a questão segurança tornou-se algo de sumo importância.

Ataques para "bisbilhotar" o tráfego na rede se intensificam, visando capturar informações importantes ou muitas vezes, tornar as informações ou suas estruturas de acesso indisponíveis. Cada serviço é disponibilizado utilizando um protocolo – ou seja, um conjunto de regras – que é associado a um número chamado porta que é onde o servidor que provê o serviço aguarda uma conexão.

Para cada serviço existe uma porta padrão. Por exemplo, 21 - FTP (transferência de arquivos), 23 - telnet (terminal virtual remoto), 25 - smtp (envio de e-mails), 80 - http (protocolo www - páginas Internet), 110 - pop3 (recebimento de mensagens). Este padrão foi estabelecido para que empresas pudessem criar suas estruturas e "conversar" com estruturas de terceiros, tornando-se um ponto favorável para toda rede de computadores. Porém, através deste padrão, invasores sabem "teoricamente" o que está sendo executado em determinada porta e podem criar seus ataques de formas bem elaboradas e com buscas muito específicas.

Enfim disso, saber quais as portas que estão abertas ou os serviços que estão sendo executados e que podem sofrer ataques devem ser avaliados. Esse artigo apresenta uma ferramenta para varreduras remotas em dispositivos utilizando a linguagem C[1][2], juntamente com a biblioteca de *Sockets*[1][2] para realização da comunicação entre a máquina a ser monitorada e a máquina a fazer o monitoramento e à paralelização para otimização do programa, a biblioteca *OpenMP*[3].

II. SOBRE O PORT SCAN DESENVOLVIDO

Um *Port Scan* nada mais é do que um programa que busca as portas abertas em um dispositivo qualquer que tenha uma interface de rede. O *Port Scan* desenvolvido foi escrito em linguagem C para execução de forma sequencial e também, em forma paralela utilizando a biblioteca *OpenMP*. O algoritmo foi escrito de forma a varrer o número máximo de portas existentes para a comunicação, ou seja, o valor 65535[2].

Para execução do programa somente foi necessária à passagem do IP ou *hostname* da máquina a ser varrida. No caso da execução com várias *threads*, foi setada a variável de ambiente OMP_NUM_*THREADS* para o valor máximo de *threads* que o programa deveria usar. Um exemplo de execução do programa é apresentado conforme mostra a figura 1.

```
#export OMP_NUM_THREADS=128; ./PortScan x.x.x.x
Escaneando portas abertas no IP\Host x.x.x.x
Porta   Servicos
21      ftp
25      smtp
22      ssh
53      domain
```

Figura 1. Exemplo de execução do *Port Scan*.

Ao executar o comando conforme ilustra a figura 1, o programa entra num *loop* para varrer as conexões abertas nas portas de 1 a 65535. Como resultado final, a figura 2 mostra a saída do programa com todos os resultados.

```
Escaneando portas abertas no IP\Host x.x.x.x
Porta   Servicos
22      ssh
21      ftp
53      domain
80      www
111     sunrpc
2049    nfs
3306    mysql
10000   webmin
37316   Servico desconhecido.
41089   Servico desconhecido.
42177   Servico desconhecido.
44621   Servico desconhecido.
49599   Servico desconhecido.
49788   Servico desconhecido.


Total de Porta(s) Open 14
Total de Porta(s) Closed 65521
Latencia do host: 0.000113/s
Tempo de execucao do programa: 0.515547/s
```

Figura 2. Amostragem dos resultados após execução.

A figura 2 mostra os resultados após uma execução com sucesso. Durante a execução do programa, para cada ocorrência de situação aberta na porta varrida, esta, é impressa automaticamente na tela. Ao fim da execução, o programa retorna a quantidade de portas abertas e fechadas encontradas, o tempo médio de latência entre todas tentativas de conexão do *Socket*[2] no dispositivo e o tempo de execução do programa. O quadro 1 mostra o trecho de código onde inicia-se o *loop*.

```
/* Trecho que inicia a paralelizacao utilizando
   a diretiva parallel for e as clausulas private
   e shared para as variaveis e schedule para o
   tamanho do bloco */
#pragma omp parallel private(dst,
                             sock,
                             portas,
                             IniLatTime,
                             FimLatTime,
                             tempoExec,
                             service)
    {
    #pragma omp for schedule(dynamic) nowait
    for(portas = 1; portas <= MaxPorts; portas++)
        {

        /* Trecho do executado dentro do laço */

        }
    }
```

Quadro 1. Trecho inicial do laço.

O quadro 1 mostra o trecho que inicia a paralelização usando a diretiva *parallel for*[3] e as cláusulas *private*[3] para indicar quais variáveis devem ter uma cópia para cada *thread* e *schedule*[3] que determina o tamanho do bloco de execuções para cada *thread*, juntamente com a cláusula *nowait*[3] que informa que tudo deve ser executado sem aguardar pelo fim de cada execução.

Como o programa busca informações de um dispositivo remoto, este calcula a latência da rede de modo que esta valha como parâmetro para análise dos resultados. O quadro 2 descreve como foi realizado esse cálculo.

```
gettimeofday( &IniLatTime, NULL );
if(connect(sock, (struct sockaddr *)&dst,
           sizeof(struct sockaddr)) == -1)
{
gettimeofday( &FimLatTime, NULL );
tempoExec = (float)(FimLatTime.tv_sec
                    - IniLatTime.tv_sec);
tempoExec += (FimLatTime.tv_usec
                    - IniLatTime.tv_usec)
                    /(float)1000000;
latencia+=tempoExec;
close(sock);
}
else
{
gettimeofday( &FimLatTime, NULL );
tempoExec = (float)(FimLatTime.tv_sec
                    - IniLatTime.tv_sec);
tempoExec += (FimLatTime.tv_usec
                    - IniLatTime.tv_usec)
                    /(float)1000000;
latencia+=tempoExec;
/* ... */
/* trecho ocultado do programa
   por não referenciar o contexto */
/* ... */
printf("Latencia do host: %.6f/s  \n",
                (latencia/65535)*100 ;
```

Quadro 2. Código para cálculo da latência.

No quadro 2, percebe-se que a variável *IniLatTime* recebe o valor inicial de tempo, conhecido através do comando *gettimeofday*[1]. Após, utilizando a biblioteca *Socket*, foi realizado um teste para verificar se o destino estava recebendo conexões naquela porta ou não. Em qualquer um dos casos, há resposta da máquina, e a variável *FimLatTime*, recebe o valor do tempo naquele instante. A latência apresentada no fim da execução do programa foi calculada através da média de todos os tempos de latência para cada teste de conexão de *Socket*.

Para mostrar o serviço que está executando em determinada porta foi utilizada a *struct servente*[2] que retorna essa informação. O quadro 3 apresenta esse trecho do código.

```
#pragma omp critical
{
    service = getservbyport(htons(portas), NULL);
    if (service>0)
        {
        printf("%d      %s\n", ntohs(service->s_port),
                               service->s_name);
        }
    else
        {
        printf("%d      Servico desconhecido.\n", portas);
        }
}
```

Quadro 3. Busca de informações da porta.

O quadro 3 apresenta um trecho de código com uma paralelização em uma área crítica, pois a impressão na tela dos valores encontrados retorna fora de ordem, devido a condição de corrida estabelecida pelo *I/O*. Através da função *getservbyport* a variável *service* recebe um *array* de informações do sistema. Caso o retorno for maior do que zero, é impresso na tela a porta e serviço nela estabelecido, caso contrário, é impresso a porta e o texto "Serviço desconhecido". A maioria das portas não possui um serviço pré-determinado por isso a função não reconhece o que está "rodando". As portas que geralmente são reconhecidas por qualquer tipo de sistema são as portas até 1024, chamadas "portas padrão", onde a maioria dos serviços conhecidos estão sendo executados.

Para o cálculo do tempo de execução o método foi semelhante ao da latência conforme explana o quadro 4.

```
main(int argc, char *argv[])
{
    gettimeofday( &IniProgTime, NULL );

    /* CODIGO DO PROGRAMA INTEIRO */

    gettimeofday( &FimProgTime, NULL );
    tempoExec = (float)(FimProgTime.tv_sec
                        - IniProgTime.tv_sec);
    tempoExec += (FimProgTime.tv_usec
                        - IniProgTime.tv_usec)
                        /(float)1000000;
    if(tempoExec>60) {
    printf("Tempo de execucao do programa:
            %.2f/m\n\n\n",tempoExec/60);
    } else {
    printf("Tempo de execucao do programa:
            %.6f/s\n\n\n",tempoExec);
    }
    return(0);
}
```

Quadro 4. Código do cálculo de execução do programa.

Como mostra o quadro 4, *IniProgTime* e *FimProgTime* são iniciadas com seus valores nos dois extremos do programa para que este seja o mais exato possível.

Como parâmetro para comparativo de desempenho foi utilizado o programa *Nmap*[4] para avaliar se o aplicativo desenvolvido gerava uma concorrência com este e se havia coerência nas informações recolhidas.

III. ESTRUTURAS, ARQUITETURAS E CONFIGURAÇÕES

Para realização dos testes foram utilizadas diversas estruturas, variando sempre a máquina que realizará a varredura, aqui denominada Monitor, e da máquina que receberá a varredura, denominada Alvo, juntamente com a estrutura de testes.

Nas arquiteturas das máquinas pode-se destacar como pontos importantes para os testes o número de *cores* existentes variando entre 1 e 8 e a velocidade da placa de rede entre 100Mb e 1Gb. É importante ressaltar que as memórias *RAM* e discos rígidos eram diferentes em cada configuração, porém, para o tipo de aplicativo desenvolvido para teste, as mesmas não interferiram nos resultados. Na estrutura da rede, tem-se testes em uma estrutura de fibra ótica, uma de par trançado e uma rede *wireless*. Referente há localização do Alvo, estes tiveram diferenças entre uma conexão ponto-a-ponto, passagem por um roteador wireless e a passagem por vários roteadores na Internet. O link de internet é de 1Mb.

As configurações de cada teste seguem apresentadas nas tabelas 1 e 2. Inicialmente a tabela 1 mostra as configurações relevantes para as análises de cada máquina monitor juntamente à topologia utilizada como conexão para os testes.

Tabela 1. Configuração das máquinas Monitoras.

Nome	Conf1	Conf2	Conf3	Conf4
Equipamento	*Netbook*	*Notebook*	ML350	ML350
Processadores	1	1	2	2
Cores	2	4	8	8
Clock	1Ghz	2.4 Ghz	2.4Ghz	2.4Ghz
Placa de Rede	100Mb	100Mb	1Gb	1Gb
Topologia	*Wireless*	Cabo TP	Fibra	Internet
S.O.	*Ubuntu* 12.04			

A tabela 2 indica as configurações dos dispositivos utilizados como Alvo no experimento.

Tabela 2. Configuração dos Alvos.

Nome	Conf1	Conf2	Conf3	Conf4
Equipamento	*Notebook*	ML110	ML350	**?**
Processadores	1	1	2	**?**
Cores	4	4	8	**?**
Clock	1Ghz	2.4 Ghz	2.4Ghz	**?**
Placa de Rede	100Mb	100Mb	1Gb	**?**
Topologia	*Wireless*	Cabo TP	Fibra	Internet
S.O.	*Ubuntu* 12.04			**?**

As escolhas destas estruturas foram feitas com o intuito de dimensionar a escalabilidade do programa em nível de *hardware* e das interferências que podem ocorrer em cada estrutura de rede.

IV. METODOLOGIAS

Os testes foram realizados executando o *Port Scan* 30 vezes em todas as configurações propostas. As variações do *Port Scan* foram feitas com execução do mesmo de forma sequencial, ou seja, sem as diretivas de paralelização; e de forma paralela utilizando 1, 2, 4, 8, 16, 32, 64 e 128 *threads*. Para cada execução foi exportada a variável de ambiente OMP_NUM_*THREADS* com o valor do número de *threads* máximo a ser executado no momento. O *Nmap* foi executado mantendo a mesma quantidade de execuções e para fins de casualidade, as execuções foram alternadas entre a sequência do *Nmap* e o *Port Scan* alternando o número de *threads*.

V. RESULTADOS E DISCUSSÕES

Inicialmente foi utilizada a ferramenta *indicator-multiload* do *Ubuntu* para gerar um gráfico de utilização da rede para cada tipo de conexão e acompanhamento para avaliação de como o programa se comporta durante a varredura, analisando, se continuamente, o mesmo fazia solicitações à máquina Alvo. A figura 3 mostra uma tela da ferramenta e o comportamento da rede numa execução.

Figura 3. Histórico dos dados recebidos e enviados na rede.

Ao visualizar a figura 3, tem-se um gráfico com os envios e recebimentos dos pacotes durante a execução do *Port Scan*. Percebe-se que o envio e recebimento formam uma simetria em vários momentos, indicando que as solicitações e as respostas são constantes durante a execução do programa.

Um fator de suma importância que será ressaltado é sobre o tempo de execução realizado nos testes.

Referente ao *Nmap* o tempo de execução é apresentado conforme gráfico 1.

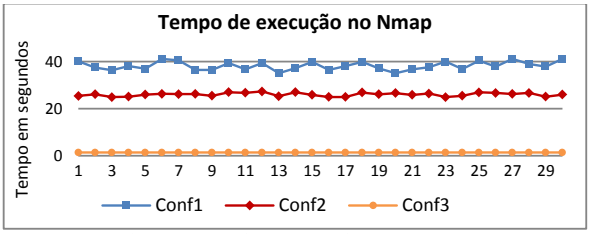

Gráfico 1. Tempo de execução dos testes no *Nmap*.

O gráfico 1 apresenta as informações de tempo de execução em cada configuração exceto a configuração 4, devido ao valor de execução ser discrepante comparado as demais medidas pelo fato da mudança de estrutura, no caso, o acesso a internet. Os valores da Conf4 serão apresentados num gráfico posterior, para fins de melhor entendimento. Quanto à linha de execução da Conf1, pode-se observar que esta, teve uma variável no tempo de execução onde, uma das causas é a estrutura, que não garante a qualidade necessária para envio e recebimento dos pacotes enviados na rede, pois uma rede *wireless* está sujeita a questões de imprevisibilidade do ambiente num todo. Outra questão é a baixa qualidade do *hardware*, que pode ter um poder de processamento inferior à demanda de requisições.

Os gráficos que seguem na sequência apresentam informações variando o programa de forma sequencial e de 1 até 128 *threads*. O gráfico 2 demonstra a média de execução na Conf1.

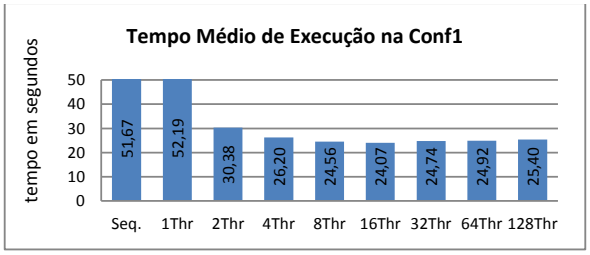

Gráfico 2. Tempo médio de execução da Conf1.

Analisando o gráfico 2, nota-se um *speedup* a partir da utilização de mais de uma *thread*. Na execução dos testes em mais de 16 *threads* percebe-se que não se obtêm uma progressão no *speedup*, devido a estrutura da Conf1.

O gráfico 3 apresenta as informações dos testes realizados na Conf2.

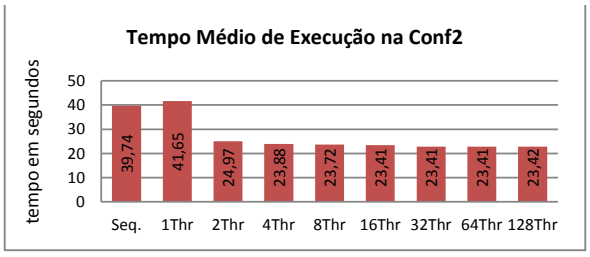

Gráfico 3. Tempo médio de execução da Conf2.

No gráfico 3, é apresentado um *speedup* no aumento do número de *threads* e após o uso de 16 *threads* é mantido o tempo de execução. Assim como na Conf1, a estrutura limita o desempenho.

No gráfico a seguir, temos os testes na Conf3.

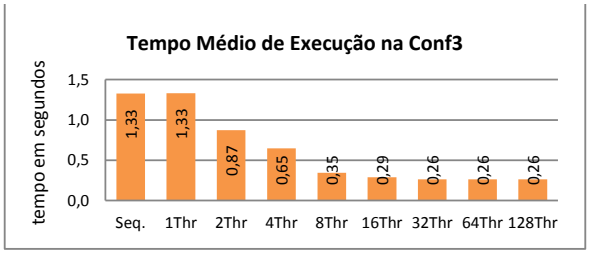

Gráfico 4. Tempo médio de execução da Conf3.

No gráfico 4, com a melhoria da estrutura, é apresentado um *speedup* no aumento do número de *threads*. Diferente das outras configurações, o gráfico se mantem após o uso de 32 *threads*, e posteriormente é mantido o tempo de execução.

Os testes realizados com a Conf4 são analisados no gráfico 5.

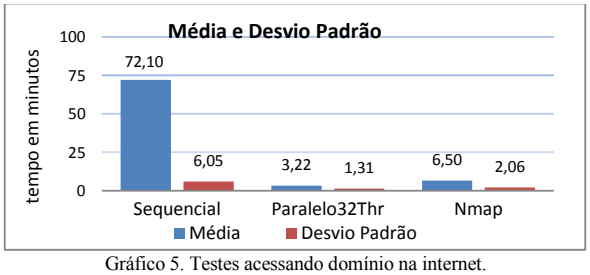

Gráfico 5. Testes acessando domínio na internet.

No gráfico 5 tem-se a comparação entre o *Port Scan* sequencial, o melhor teste de paralelização do mesmo, onde foram utilizadas 32 *threads*, e o *Nmap*. Inicialmente, pode-se notar que o resultado apresenta uma grande diferença na média da execução do *Port Scan* de forma sequencial para a forma paralela, onde a melhor forma paralela chegou a ser 22 vezes mais rápida na média geral.

O programa paralelo também teve melhores resultados quanto à comparação com o *Nmap*, sendo em média 2 vezes mais rápido. É válido salientar os tempos mínimos e máximos de execução de cada teste, onde o programa sequencial teve como tempo mínimo 61,07m e máximo de 83,23m, o paralelo 2,41m e 5,12m e o *Nmap* 4,23m e 9,12m.

É adequado observar que em todas as execuções contra os Alvos, o *Port Scan* identificou a mesma quantidade e as mesmas portas encontradas em aberto, tanto nos Conf*(n)*, quanto nas variações de número de *threads*, ou às comparações com o *Nmap*.

VI. Conclusões

Com esta implementação conclui-se que ao empregar paralelização utilizando *OpenMP* para o *Port Scan* desenvolvido, o mesmo obteve um amplo *speedup* devido ao aproveitamento das quantidades de *cores* em cada Monitor testado, juntamente com o aumento do mesmo quando alternadas as quantidades de *threads* por execução.

A quantidade de testes e a diversidades dos ambientes e configurações demonstraram bons resultados no comportamento do mesmo quanto à sua escalabilidade. As comparações com o programa *Nmap* apresentaram que o mesmo foi em média duas vezes superior em tempo de execução. Não é possível afirmar o porquê desta melhoria nesta comparação, pois as variáveis a serem discutidas são muitas, como por exemplo, é desconhecido o número de testes que o *Nmap* faz para cada conexão em determinada porta para definir seu *status*. Talvez, o mesmo faça um maior número, o que pode influenciar gravemente no tempo de execução.

Percebeu-se nos testes da Conf4 que a falta de conhecimento da estrutura Alvo – tais como, link, processamento, demanda/número de conexões, dentre outros – influenciaram muito no tempo de execução de todos os testes.

Como trabalho futuro incentiva-se maior estudo da biblioteca *Sockets* e a biblioteca *OpenMP*, pois, possivelmente, melhores soluções podem surgir juntamente com a busca de mais informações relevantes a implementação e monitoramento para segurança de dispositivos, computadores e servidores numa rede.

Referências

[1] W. R. Stevens, "Unix Network Programming," Interprocess Communications, vol. 2, Prentice Hall, 1998.

[2] W. R. Stevens. B. Fenner. A. M. Rudoff, "Programação de Rede UNIX," API para Soquetes de Rede, vol. 1, Artmed, 2005.

[3] OpenMP. (2012, 2 April). Site oficial da API [Online]. Available: http://openmp.org.

[4] Nmap. (2012, 7 January). Site oficial do Aplicativo [Online]. Available: http://nmap.org/book/man.html.