Monografia

Análise Comparativa de Desempenho de um algoritmo Paralelo implementado nas Linguagens de Programação CPAR e OpenMP

disciplina: Programação Paralela e Distribuída

Professora: Liria Matsumoto Sato

Autores
Claudio Penasio Junior
Leonardo Mattes

São Paulo - 2004

Sumário

1 Introdução....3
2 Objetivo....3
3 Linguagem de Programação Paralela....3
3.1 A Linguagem de Programação CPAR....3
3.2 A linguagem de Programação OpenMP 10
Construções Sections 11
Flush....13
4 O algoritmo....14
4.1 A implementação seqüencial....14
4.2 A implementação em Cpar....15
4.3 A implementação em OpenMp....16
5 O Cenário de testes....17
6 Resultados obtidos....17
7 Conclusão....18
8 Referências Bibliográficas....20

1 Introdução

Este trabalho possui dois principais objetivos, que são apresentar um breve resumo comparativo entre as linguagens de programação paralelas CPAR (Linguagem de Programação Paralela, desenvolvida na Escola Politécnica da Universidade de São Paulo) e o OpenMP (API - Application Program Interface para paralelismo multi-thread em sistemas baseados em memória compartilhada desenvolvido e mantido pelo OpenMP ARB - OpenMP Architecture Review Board. E avaliar o desempenho e o comportamento de um mesmo algoritmo escolhido nas as duas linguagens.

2 Objetivo

O Objetivo deste trabalho é traçar um paralelo básico entre as Linguagens de Programação Paralela CPAR e OpenMp, bem como avaliar o desempenho e o comportamento de um algoritmo escolhido nas as duas linguagens.

3 Linguagem de Programação Paralela

As linguagens de programação paralela surgiram como um novo recurso para facilitar a tarefa de construir programas que utilizem os recursos de paralelismo de equipamentos com arquitetura de computação paralela. Para utilizar plenamente os recursos de um equipamento multiprocessado, existem basicamente duas opções, construir programas explicitando e controlando manualmente os trechos de paralelismo, utilizando recursos de uma Linguagem de Programação como o C para a criação das threads e processos necessários, ou utilizar-se de uma linguagem de programação paralela, que na verdade normalmente não passam de diretivas de compilador para uma linguagem de programação pré existente como por exemplo C, que possuem recursos que facilitam o tratamento de trechos paralelos e seqüenciais dos programas, atuando como um pré-compilador e criando a versão final em uma linguagem de programação como C, poupando ao programador a tarefa de se preocupar com o gerenciamento de threads e processos. Desde o surgimento dessas arquiteturas, surgiram diversas linguagens de programação paralela, entre elas a CPAR criada pela Professora Doutora Liria Matsumoto Sato da Escola Politécnica da Universidade de São Paulo e a OpenMP foi desenvolvido por um grupo composto pelos maiores fabricantes de software e hardware do mundo. O desenvolvimento é realizado através de um trabalho colaborativo entre os diversos parceiros interessados no projeto, entre eles, estão: Compaq/Digital, Hewlett-Packard, Intel, IBM, KAI, Silicon Graphics, Sun entre outros. Tanto a CPAR quanto a OpenMP, são compostas de três elementos básicos que são: Diretivas de Compilador, Biblioteca de Rotinas e Variáveis de Ambiente.

3.1 A Linguagem de Programação CPAR

A linguagem CPAR é uma extensão da linguagem C, na qual foram acrescentadas construções para expressar o paralelismo. Na CPAR, um bloco de tarefas seqüenciais

pode ser executado paralelamente a outros blocos de tarefas seqüenciais. Para explorar tanto o paralelismo em níveis de granularidade grossa como fina, com relação às subrotinas a CPAR utiliza uma abordagem de microtarefa e macrotarefa. Em cada macrotarefa podem haver inúmeras microtarefas, que são executadas paralelamente. A CPAR também possui recursos para explorar a hierarquia de memória compartilhada

com o uso de variáveis compartilhadas, que podem ser locais ou globais a macrotarefa.

Modelos de Programação:

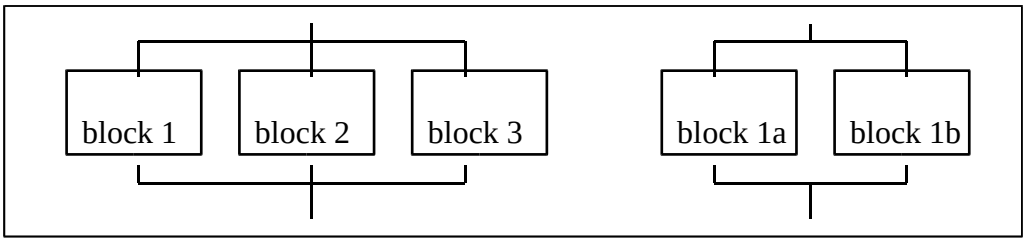


Figura 1 Processo Principal com múltiplos níveis de paralelismo

Tornar um programa paralelo é distribuir se trabalho entre os processadores disponíveis. O modelo apresentado pela CPAR permite o uso de múltiplos níveis de paralelismo.

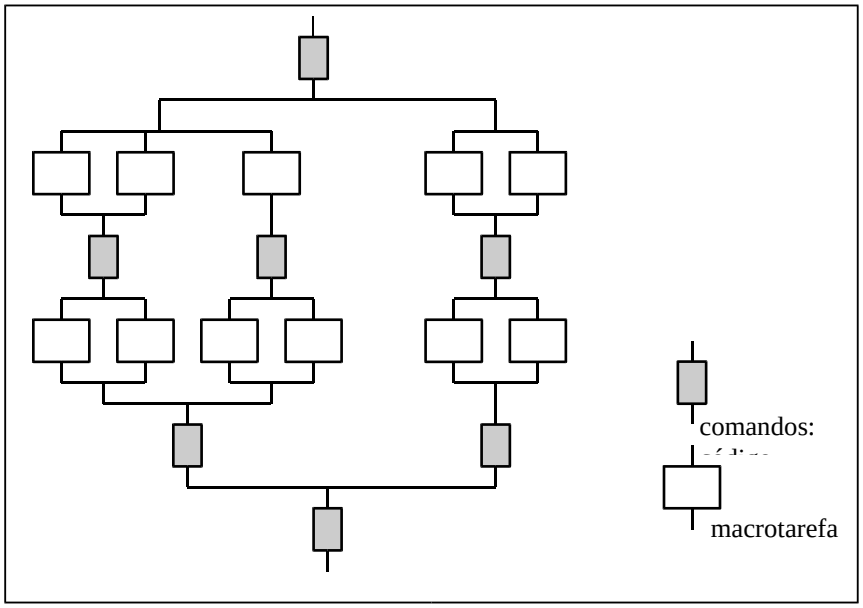


Figura 2O modelo de Programação da CPAR, evidenciando os níveis de paralelismo

Elementos do Modelo de Programação:

- **elemento seqüencial:** é uma porção de comandos, que deve ser executada seqüencialmente;
- **macrotarefa:** é uma porção de código, ao nível de subrotina, que pode conter um nível mais fino de paralelismo, as microtarefas. Macrotarefas podem ser executadas simultaneamente;

  Declaração da macrotarefa:

```
task spec nome_da_macrotarefa(parâmetros)
{ declarações }
```

  Declaração do corpo da macrotarefa:

```
task body spec nome_da_macrotarefa(parâmetros)
declaração de parâmetros
{
        declarações de variáveis
        comandos
}
```

  Criação da macrotarefa:

```
create n, nome_da_macrotarefa(argumentos)
```

Onde n é o número de processadores para executar a macrotarefa.

- **microtarefa:** é uma porção de código seqüencial, contida em um laço, cujas iterações são executadas paralelamente, ou em um bloco de comandos que é executado paralelamente a outros blocos;

  Em uma microtarefa podemos encontrar dois tipos de paralelismo, o paralelismo homogêneo e o heterogêneo. O paralelismo homogêneo pode ser utilizado através do comando *forall,* enquanto que o paralelismo heterogêneo pode ser utilizado através do comando *parbegin.*

  O comando *forall:*

```
forall i=1 to max
{
        A[i] = i+1;
}
```

O comando *parbegin:*

```
parbegin
        a=b*3;
also
        a=b+sqr(a);
parend
```

- **blocos paralelos:** são porções do código da função principal (“main”), do programa que são executadas paralelamente. Um bloco pode conter blocos paralelos, ou seja é permitido o aninhamento de blocos. Um bloco pode conter elementos seqüenciais, macrotarefas e blocos paralelos. Para blocos paralelos utilizamos o comando cobegin.

  O comando *cobegin*:

```
cobegin
        a=b*3;
also
        a=b+sqr(a);
also
        ...
also
        ...
coend
```

- **função principal:** é a função principal do programa (“main”). Pode conter blocos, elementos seqüenciais e macrotarefas.

Variáveis:

A CPAR permite a declaração de variáveis locais, globais compartilhadas e locais

compartilhadas. Para declararmos uma variável compartilhada utilizamos a diretiva *shared.*

```
shared tipo nome_da_variável;
```

Sincronização:

Em uma linguagem de programação paralela é importante que se tenha disponível elementos de exclusão mútua para garantir utilização de memória compartilhada segura, na CPAR os mecanismos que permitem a exclusão mútua são: os monitores, os semáforos, e os eventos.

Monitores:

Os monitores são uma forma de acesso seguro a variáveis compartilhadas através de funções que são executadas com exclusividade . O acesso é possível apenas através das funções do monitor.

Declaração do monitor:

```
monitor nome_do_monitor
        declaração de variáveis compartilhadas
        declaração de variáveis locais
        {
                funções do monitor
        }
        {
                iniciação do monitor
        }
```

Chamada do monitor:

```
nome_do_monitor.função(argumentos);
```

Semáforos:

O semáforo é uma construção que permite tanto o incremento quanto o decremento atômico de variáveis de sincronização. Quando uma tarefa obtém o controle do semáforo ele decrementa seu valor (operação P), e quando a tarefa libera o semáforo o seu valor é novamente incrementado (operação V). Um semáforo só pode ser obtido quando atinge valores positvos.

Definição de um semáforo:

```
shared Semaph nome_do_semáforo;
```

O semáforo possui as seguintes funções: criação, remoção, obtenção e liberação.

Funções do semáforo:

```
create_sem(&nome_do_semáforo,valor_inicial);
rem_sem(&nome_do_semáforo);
lock(&nome_do_semáforo);
unlock(&nome_do_semáforo);
```

Eventos:

O evento é uma construção utilizada para evitar condições de “deadlock”, uma vez que permite que um processo espere por um determinado “evento” ativado por outro processo .

Declaração de um evento:

```
shared Event  nome_do_evento;
```

Em relação aos eventos existem as seguintes operações: criar, ativar, remover, apagar,ler seu estado e esperar pelo seu estado.

Operações do evento:

```
create_ev(&nome_do_evento);
set_ev(&nome_do_evento);
rem_ev(&nome_do_evento);
res_ev(&nome_do_evento);
int read_ev(&nome_do_evento);
wait_ev(&nome_do_evento);
```

Passagem de Mensagens:

Para a passagem de mensagens em CPAR é necessário primeiro se criar uma “entrada”, para em seguida poder utilizar o envio, recebimento ou verificação.

Declaração de uma entrada:

```
entry nome_da_entrada(declaração do tipo da mensagem);
```

Transmissão da mensagem:

```
send  nome da macrotarefa.nome_da_entrada(mensagem);
```

Recepção da mensagem:

```
receive nome_da_entrada(variável);
```

Verificação da chegada da mensagem:

```
state nome_da_entrada(variável);
```

3.2 A linguagem de Programação OpenMP

A "OpenMP" oferece um conjunto de diretivas para a programação paralela em sistemas multiprocessados com memória compartilhada, realizando para tal a criação e o controle de Threads. As diretivas da OpenMP podem ser incluídas em programas escritos nas linguagens “C” e FORTRAN para especificar pedaços de programas que devem ser executados em paralelo.

Para a implementação dos laços paralelos a OpenMP utiliza a biblioteca "Pthreads", o que que lhe proporciona um bom desempenho e um alto grau de portabilidade, uma vez que "Pthreads" possui desempenho otimizado e já foi implementada em uma inúmeras de plataformas.

Internamente a OPENMP trabalha com um modelo “FORK-Join” onde a partir de uma “thread master” (aquela que iniciou o programa) cria-se um time de threads para a realização de tarefas em paralelo, e depois é realizado um join onde todas as *threads* são sincronizadas e destruídas, sobrando somente a master thread.

Modelo Fork - Join [6]:

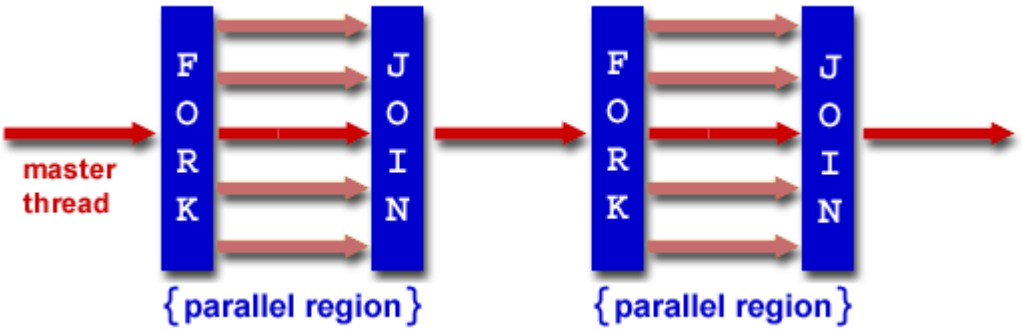


Figura 3O modelo de threads fork- join

Diretivas da OpenMP

A OpenMP conta com diretivas para definir e controlar um pedaço do programa a ser executado em paralelo, a seguir discutiremos o funcionamento de alguma diretivas:

Construções paralelas

A diretiva para construção paralela #pragma omp parallel permite ao programador definir um pedaço do programa que será executado em paralelo por threads, definindo vaiáveis que serão mantidas de forma compartilhadas

Os parâmetros acima representa diferentes usos de lista de variáveis :

```
#pragma omp for shared(lista) private(lista) firstprivate(lista)
copyin(lista) redution(lista) [nowait]
   for (...){
    }
```

- **shared** - são variáveis compartilhadas por todas as Threads;
- **privates** - são variáveis são privadas as threads;
- **firstprivate -** não são compartilhadas porém são iniciadas fora da thread;
- **copyin -** as variáveis possuem o valor inicial igual ao valor da thread principal;
- **redution** - define uma formula para combinar as variáveis de todas threads nas variável da thread master;

Construções *for/do*

A construção for permite com que um instrução do tipo laço *for* ou *do sejam* realizadas em paralelo por threads, definindo uma relação de variáveis


```
#pragma omp for shared(lista) private(lista) firstprivate(lista)
copyin(lista) redution(lista) [nowait]
   for (...){
    }
```


privadas e compartilhadas pelas threads, segue formato de uma construção for:

O parâmetro *nowait* define se deve-se esperar a finalização de todas threads.

Construções Sections

A diretiva *section* permite definir seções de código distintas que serão executadas em paralelos (paralelismo heterogêneo), conta com o seguinte formato:

```
#pragma omp sections [clause ...]  newline
        shared (list)
            private (list)
            firstprivate (list)
            lastprivate (list)
            reduction (operator: list)
            nowait
 {
```

```
  #pragma omp section   newline
    structured_block1
  #pragma omp section   newline
    structured_block2
}
```

No exemplo acima as bloco de instruções “block1” e “ bloc” serão executados em paralelo, podendo possuir uma lista de variáveis compartilhadas *shared* e privadas *private.*

Construções de sincronizações

Uma vez que a execução de uma *thread* pode depender do resultado de outra, a sincronização tem como objetivo ordenar a execução das *threads* para o correto funcionamento do programa. A OpenMP apresenta as seguintes diretivas de para sincronizar threads: singlem; master; critical; atomic; order; barrier; flush.

Master

A diretiva *master* define uma região que será executada somente pela *thread master*,

```
#pragma omp master  newline
  structured_block
```

sendo tal região ignorada pelas outras threads, a figura abaixo demonstra seu formato

Critical

Define uma região critica que poderá ser executada somente por uma *thread por vez*, bloqueando o acesso de outras *threads*, a figura abaixo demonstra seu formato :

```
#pragma omp critical
  structured_block
```

Atomic

Define que a atribuição de uma variável será realizada somente por um thread de cada vez impedindo o acesso simultâneo de outras threads ao mesmo temo, a figura abaixo demonstra seu formato :

```
#pragma omp atomic
  var op = <expresão
```

Ordered

A diretiva *ordered* define que as interações em um determinado *loop* serão executados ao mesmo tempo como em um programa serial, a figura abaixo demonstra seu formato :

```
pragma omp ordered  newline

  structured_block
```

Barrier

A diretriz *Barier* sincroniza todas *thread* correntes esperando a finalização de todas para retornar somente a execução da *thread master*, a figura abaixo demonstra seu formato :

```
pragma omp barrier   newline
```

Flush

A diretiva *flush* tem como funcionalidade sincronizar o valor das variáveis em paralela de forma que todas as threads tenham acesso aos valores atualizados e corretos das

```
#pragma omp flush (list)  newline
```

variáveis compartilhadas, a figura abaixo demonstra seu formato:

4 O algoritmo

O algoritmo escolhido foi um algoritmo de multiplicação de matrizes, por uma maneira de implementação bem conhecida e generalista, causando um grande esforço computacional.

4.1 A implementação seqüencial

```
/*Multiplicação de matrizes sequencial*/
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "mede_time.h"

#define SIZE 1000

float A[SIZE][SIZE];
float B[SIZE][SIZE];
float C[SIZE][SIZE];
void inicializa_matriz(){
      int i,j;
      for (i=0;i < SIZE;i++){
            for(j=0;j<SIZE;j++){
                  A[i][j]=3*i+j;
                  B[i][j]=i+3*j;
                  C[i][j]=0.0;
            }}}
void multiplica_matriz(){
      int i,j,k;
      for (i=0;i < SIZE;i++){
                  for(k=0;k<SIZE;k++){
                    for(j=0;j<SIZE;j++){
                        C[i][j]=C[i][j]+A[i][k]*B[k][j];
                  }}}}
main(){
      inicializa_matriz();

      TIMER_CLEAR;
      TIMER_START;
       multiplica_matriz();
      TIMER_STOP;
            printf ("TEMPO [%d]: %12.7f\n",SIZE,TIMER_ELAPSED );
}
```

4.2 A implementação em Cpar

```
/*Multiplicação de matrizes cpar*/
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "mede_time.h"

shared int size;
shared float A[1000][1000];
shared float B[1000][1000];
shared float C[1000][1000];

task spec inicializa_matriz();
task body inicializa_matriz(){
      int i,j;
      size=1000;
      forall i=0 to size-1 {
            for(j=0;j<size;j++){
                  A[i][j]=3*i+j;
                  B[i][j]=i+3*j;
                  C[i][j]=0.0;
            }}}
task spec multiplica_matriz();
task body multiplica_matriz(){
      int i,j,k;
      size=1000;
      forall i=0 to size-1 {
            for(k=0;k<size;k++){
              for(j=0;j<size;j++){
                        C[i][j]=C[i][j]+A[i][k]*B[k][j];
                  }}}}

main(){
      int nprocs=4;
      alloc_proc(nprocs);
      create nprocs, inicializa_matriz();
      wait_all();
      TIMER_CLEAR;
      TIMER_START;
       create nprocs, multiplica_matriz();
       wait_all();
      TIMER_STOP;
            printf ("TEMPO [%d]: %12.7f\n",size,TIMER_ELAPSED );


}
```

4.3 A implementação em OpenMp

```
/*Multiplicaçã de matrizes OpenMP */
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include "mede_time.h"


#define SIZE 1000
int size;
float A[SIZE][SIZE];
float B[SIZE][SIZE];
float C[SIZE][SIZE];

main()
{
int nproc;
int i, j, k, n;

size = SIZE;
nproc =2;

#pragma omp parallel shared(A, B, C, size) private(i,j, k)
{
   #pragma omp for
   for(i=0; i <size; i++)
     { for(j=0;j<size;j++)
        {A[i][j]=3*i+j;
           B[i][j]=i+3*j;
         C[i][j]=0.0;}
     }

    TIMER_CLEAR;
    TIMER_START;

  #pragma omp for
  for(i=0; i<size; i++)
    {for(k=0;k<size;k++)
        {for(j=0;j<size;j++)
            {C[i][j]=C[i][j]+A[i][k]*B[k][j];}
        }
     }
      TIMER_STOP;
           printf ("TEMPO [%d]: %12.7f\n",size,TIMER_ELAPSED );


}

 printf (" FIm ");

}
```

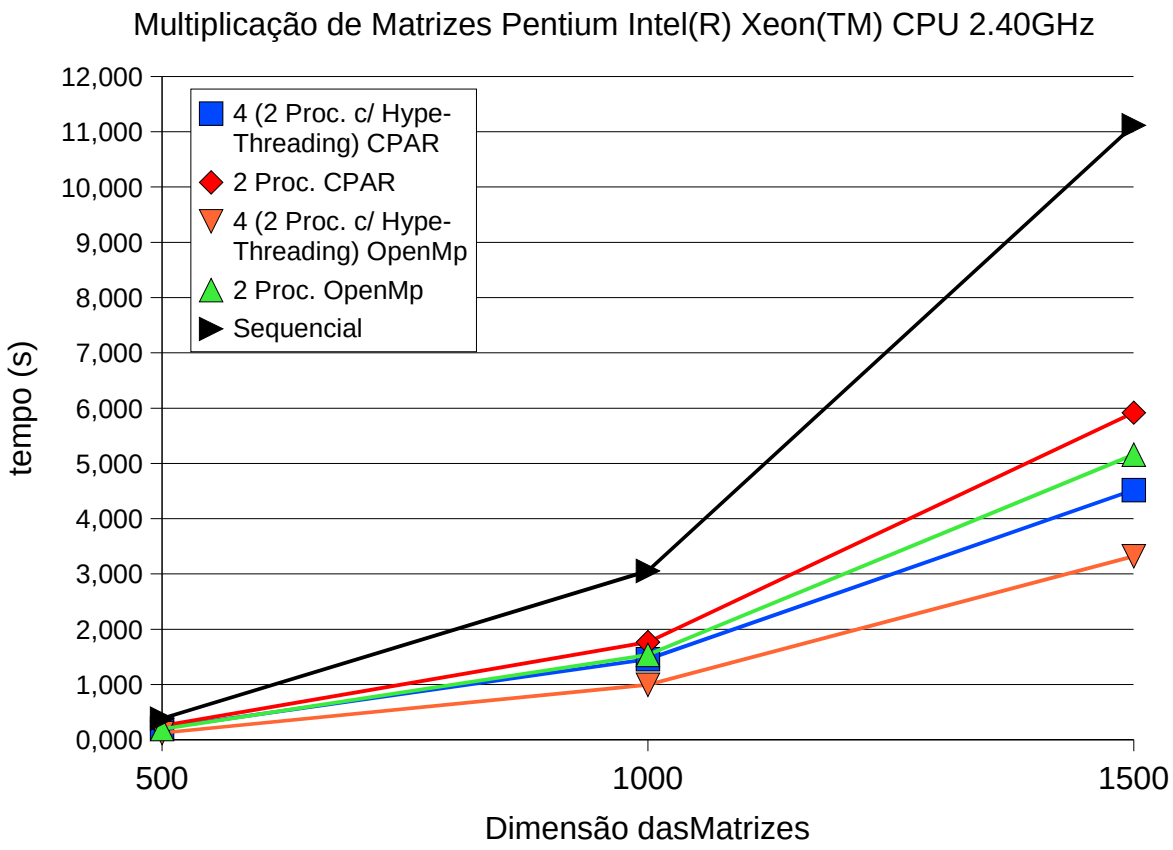
5 O Cenário de testes

Os testes foram executados em uma máquina com 2 processadores Intel(R) Xeon(TM) CPU 2.40GHz com a possibilidade de configuração para Hyperthread, está máquina possui 1000 MB de memória RAM e possui um HD do tipo SCSI de 36.7 GB. O Sistema Operacional instalado é o Red Hat Linux 9.0 com kernel 2.4.20-8smp

6 Resultados obtidos

Matrizes	4 (2 Proc. c/ Hyper-Threading) CPAR	2 Proc. CPAR	4 (2 Proc. c/ Hyper-Threading) OpenMp	2 Proc. OpenMp	Sequencial
500	0,214	0,249	0,123	0,194	0,384
1000	1,457	1,765	0,996	1,536	3,054
1500	4,518	5,918	3,320	5,157	11,114
2000	*	*	7,990	12,833	24,402

*Com Matrizes de tamanho 2000 o CPAR deu o seguinte erro na hora da execução:Erro na criação de memória compartilhada,Tamanho a ser alocado: 48000008, Tamanho invalido!,Erro na utilização de memória compartilhada,Tamanho invalido!



7 Conclusão

Diante dos testes realizados, pudemos observar que sem dúvida a paralelização de programas reflete em um significativo ganho de performance na realização da mesma tarefa, que no caso foi a multiplicação de matrizes. A observação da tabela acima nos mostra que no caso comparativo de uma aplicação seqüencial e uma aplicação paralela, o ganho de performance variou de 36% no pior caso (codificação em CPAR usando 2 processadores) e de 68% no melhor caso (codificação em OpenMP usando 4 - 2 processadores com Hyper-Threading) para multiplicação de matrizes de dimensões 500x500. E um ganho de 46% no pior caso (codificação em CPAR usando 2 processadores) e de 70% no melhor caso (codificação em OpenMP usando 4 - 2 processadores com Hyper-Threading) para multiplicação de matrizes de dimensões 1500x1500. Já a comparação direta de aplicações paralelizadas entre a linguagem de programação CPAR e a OpenMP, nos mostra que no caso mais próximo o ganho de performance foi de 13% utilizando OpenMp (2 processadores com matrizes de dimensões 1500x1500), e no caso mais distante o ganho de performance foi de 42% utilizando OpenMp (4 – 2 processadores com Hyper-Threading com matrizes de dimensões 500x500). Observando os resultados acima, observa-se que a linguagem OpenMp tem como ponto forte no seu ganho de performance a utilização de Threads, recurso este que não havia sido implementado na versão de produção do CPAR utilizada no teste, porém conforme [5] já está disponível sua implementação em versões de teste. Um outro ponto importante é que a linguagem de programação CPAR possui sua atualização e ciclo de desenvolvimento restrito ao programa de graduação e pós graduação da EPUSP, fator

que sem dúvida acarreta uma morosidade tanto na implementação de novos recursos, quanto ao que se refere ao teste da linguagem na procura por possíveis “bugs”. Uma sugestão importante que poderia funcionar como um grande impulso para o CPAR seria colocar seu código sob uma licença de código aberto, como por exemplo o GPL da Free Software Foundation http://www.fsf.org , isso faria com que um número muito maior de pessoas realizassem testes, e o desenvolvimento continuaria sob coordenação da EPUSP porém com colaboração da comunidade “opensource”. A linguagem CPAR apesar de ter apresentado um desempenho inferior nos testes realizados, é sem dúvida uma linguagem com um apelo didático muito forte, uma vez que sua estrutura faz com que o estudante de programação paralela visualize de forma muito mais precisa as características importantes do ato de paralelizar aplicações de forma eficiente.

8 Referências Bibliográficas

[1] Kernigham, Brian W; Ritchie, Dennis M.; “The C Programming Language”, PRENTICE HALL, 1978.

[2] Tanenbaum, Andrew S.; “Modern Operating Systems”, PRENTICE HALL, 1992.

[3] Ben-ari, M. "Principles of CONCURRENT and DISTRIBUTED PROGRAMMING", PRENTICE HALL, 1990.

[4] Sato, Liria M.; Knop, Felipe; Midorikawa, Edison T.; Bernal, Volnys B.; Yu, Wang K.; “Uma Introdução ao Software Básico de Sistemas Multiprocessadores”, 1991.

[5] Matheos Junior, Walter; “ Uma Implementação da Linguagem CPAR usando um modelo de Programação 'Threads'”, 2002.

[6] http://www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html#Synchronization