Carlos Eduardo Benevides Bezerra

Disciplina: CMP134 – Introdução ao processamento paralelo e distribuído
Prof.: Nicolas Maillard e Philippe Navaux

Relatório do trabalho desenvolvido:
Uma comparação entre três diferentes API's de programação paralela

Introdução

Desde o surgimento dos processadores superescalares e, mais recentement, das arquiteturas *multicore* (vários núcleos de processamento no mesmo processador), têm sido criadas diversas ferramentas que possibilitam ao desenvolvedor de software criar aplicações que exploram explicitamente o paralelismo do processamento. Dois exemplos disso são as especificações do OpenMP e do MPI.

Cabe, então, sempre que surge uma nova funcionalidade nessas abordagens de programação paralela, fazer estudos que avaliem seu desempenho, não necessariamente classificando-as em boa ou ruim, mas indicando para qual situação cada uma é mais indicada.

Neste trabalho, é feita uma comparação entre as especificações do MPI, do OpenMP 2.5 e do OpenMP 3.0. São feitas três implementações diferentes de um algoritmo de cálculo de produto matricial, cada uma explorando algumas características de cada uma destas APIs.

São encontrados alguns resultados, evidenciando que, para uma arquitetura de memória compartilhada, mesmo que o MPI oferece uma maior liberdade ao programador, é muito provável que uma aplicação programada com o OpenMP tenha melhor desempenho do que outra, programada com envio e recebimento de mensagens (MPI).

O texto está organizado da seguinte forma: primeiramente, é descrita de maneira breve cada possível implementação seqüencial do cálculo de produto matricial. Em seguida, é apresentado cada algoritmo paralelo implementado, fazendo também uma rápida descrição do conceito de **tasks**, introduzido no OpenMP 3.0. Por último, são apresentados os resultados das simulações e as conclusões a que se chegou neste trabalho.

Multiplicação de matrizes

O problema de calcular o produto de duas matrizes de ordem n é trivial, porém permite avaliar de maneira simples diferentes abordagens para programação da solução. O produto matricial já foi largamente estudado, tendo como algumas de suas soluções mais importantes:

- Algoritmo clássico iterativo, com três laços aninhados: $O(n^3)$;
- Algoritmo clássico recursivo: $O(n^3)$;
- Algoritmo recursivo de Strassen: $O(n^{2,8})$ e
- Algoritmo de Coppersmith-Winograd: $O(n^{2,376})$.

Como não é o objetivo deste trabalho reduzir ao máximo o tempo de execução, mas verificar a diferença de desempenho de algoritmos paralelizados com diferentes APIs, optou-se por simular as versões iterativa e recursiva clássicas do produto matricial.

O algoritmo iterativo para cálculo do produto matricial das matrizes A_{nxn} e B_{nxn} – retornando o resultado na matriz C_{nxn} – bastante simples:

```
Para i:=1 até n faça
     Para j:=1 até n faça
            C[i,j] := 0;
            Para k:=1 até n faça
                 C[i,j] += A[i,k]*B[k,j];
            Fim Para
     Fim Para
Fim Para
```

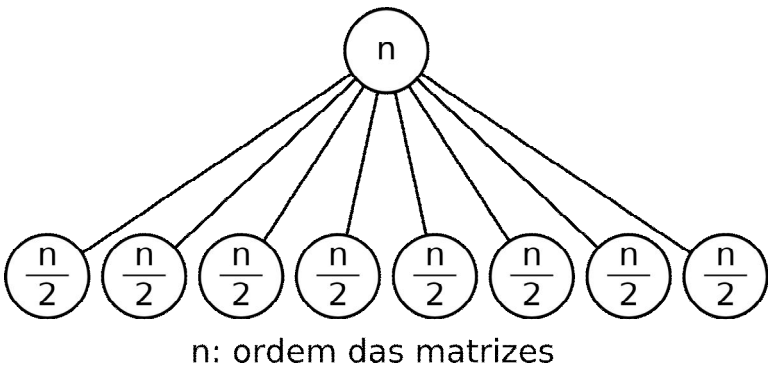
Este algoritmo, cuja complexidade é $O(n^3)$, é trivial de se paralelizar com OpenMP, mesmo que o valor de **n** não seja uma constante definida em tempo de compilação e, conseqüentemente, não se possa utilizar o **`#pragma omp parallel for`**.

A versão recursiva da multiplicação de matrizes, no entanto, requer que cada matriz seja sub-dividida em quatro sub-matrizes. A partir de operações de multiplicação dessas sub-matrizes, obtém-se a matriz resultante. Porém, essas multiplicações chamadas recursivas ao próprio algoritmo de multiplicação sendo utilizado.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$

Cada nodo da árvore de recursão dessa abordagem tem, então, oito filhos, cada um representando uma multiplicação de duas matrizes de ordem n/2. Tem-se a complexidade de: $T(n) = 8*T(n/2) + 4*(n/2)^2$. Pelo teorema mestre: $T(n) = O(n^3)$.



n: ordem das matrizes

Implementações paralelas

- OpenMP 2.5: algoritmo iterativo

```
matriz mult (matriz A, matriz B) {
     int ordem, x,y;
     matriz C;
     ordem = A.ordem;
     C = new_matrix(ordem);

     #pragma omp parallel
     {
          int i,j,k;
          int tid = omp_get_thread_num();
          int chunk_size = ordem / omp_get_num_threads();
          if (ordem % omp_get_num_threads()) chunk_size++;
          int chunk_start = tid * chunk_size;
          int chunk_end = chunk_start + chunk_size;
          for (i = chunk_start ; i < chunk_end && i < ordem ; i++)
          {
               for (j = 0 ; j < ordem ; j++) {
                    E(C, i, j) = 0;
                    for (k = 0 ; k < ordem ; k++) {
                         E(C, i, j) += E(A, i, k) * E(B, k, j);
                    }
               }
          }
     }
     return C;
}
```

Nesta implementação, é calculada uma faixa de índices do laço mais externo que será calculada por cada uma das threads. No caso da divisão não ser exata, arredonda-se o valor para cima, verificando na condição do laço se o índice ultrapassou a faixa de valores correta. Cada thread, então, executa apenas sua faixa de iterações. Como cada passo do laço mais externo é independente e não há escrita concorrente à mesma área de memória nesta implementação, não é necessário utilizar algum mecanismo de sincronização entre as threads..

- OpenMP 3.0: algoritmo recursivo (uso de tasks)

```
{
     //(...)

     matriz A_11, A_12, A_21, A_22;
     matriz M_111, M_112, ..., M_222;
     part_matrix (A, &A_11, &A_12, &A_21, &A_22);

     //(...)

     #pragma omp task shared(M_111)
     M_111 = recursive_mult (A_11, B_11);
```

```
        #pragma omp task shared(M_112)
        M_112 = recursive_mult (A_12, B_21);

        //(...)

        #pragma omp task shared(M_222)
        M_222 = recursive_mult (A_22, B_12);

        #pragma omp taskwait

        matriz C_11 = some (M_111, M_112);
        matriz C_12 = some (M_121, M_122);
        matriz C_21 = some (M_211, M_212);
        matriz C_22 = some (M_221, M_222);

        C = merge_matrix(C_11, C_12, C_21, C_22);

        return C;
}
```

Nesta implementação, é utilizada uma característica da versão 3.0 do OpenMP, que são as **tasks**. *Tasks* são blocos de código que podem ser executados em paralelo enquanto outra thread avança para o código após esse bloco. Quando uma thread está executando uma determinada função, por exemplo, e encontra essa primitiva, ela delega a task para outra thread e avança na execução, no código após a task.

Para manter a sincronização das diferentes threads, após as inserções das diretivas **#pragma omp task**, põe-se uma diretiva **#pragma omp taskwait**. Ao se chegar nessa diretiva a thread principal (daquele bloco de código) pára e espera que as tasks que ela criou retornem, para então continuar a execução. Seu funcionamento é bastante parecido com o do **#pragma omp barrier**.

Uma característica muito importante do uso de tasks é o fato de que é necessário definir quais as variáveis que serão compartilhadas pelas threads executando as tasks. Embora no OpenMP 2.5 as variáveis instanciadas fora do bloco paralelo fossem compartilhadas por *default*, no caso das tasks é necessário especificar quais serão essas variáveis compartilhadas. Isso depende de cada tipo de variável No nosso caso, como são variáveis no escopo de uma função, o OpenMP 3.0 não as compartilha por padrão.

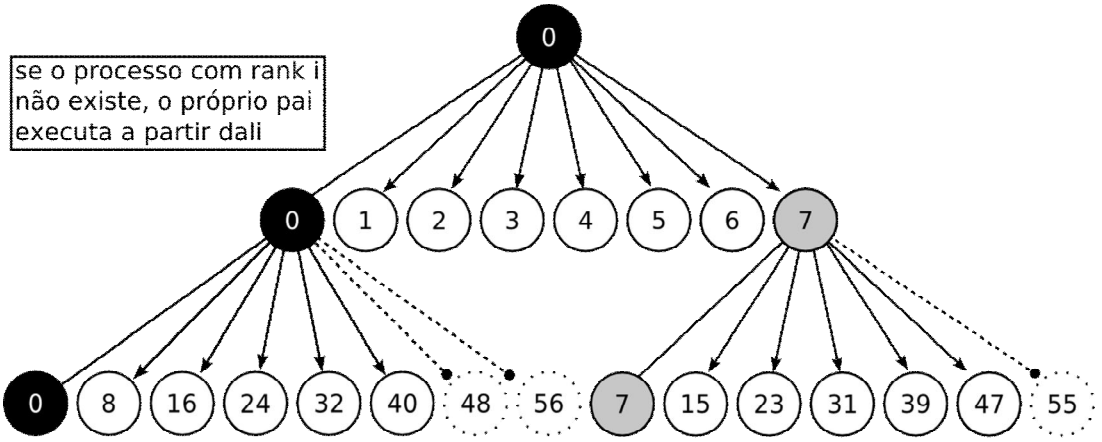
Por último, o escalonamento das tasks para serem executadas pelas threads é dinâmico, ou seja, as tasks não estam estaticamente associadas cada uma a uma thread. Ao contrário, quando houver qualquer thread ociosa, esta irá executar a task pendente.

O conceito de tasks veio para facilitar a paralelização de algoritmos irregulares, que não estão dentro de um laço **`for`** bem definido, por exemplo, e, principalmente, para facilitar a paralelização de algoritmos recursivos, o que era mais complicado de se fazer em versões anteriores da API do OpenMP.

No caso do algoritmo recursivo de multiplicação de matrizes, cada chamada recursiva pode ser executada em paralelo, enquanto o restante do código da função continua executando, bastando apenas ter um ponto de barreira antes de utilizar os resultados das chamadas recursivas. Por este motivo, cada chamada recursiva está definida como uma

task independente, a ser executada em paralelo e, antes de somar os resultados das multiplicações, é inserido um ponto de sincronização, para que todos os resultados estejam disponíveis antes de serem utilizados.

- MPI: algoritmo recursivo, com espera e despacho de ordens de execução



Exemplo com 48 processos
(rank 0 a 47)

O algoritmo recursivo implementado em MPI funciona da seguinte: o processo de rank 0 começa a recursão, distribuindo cada uma das multiplicações recursivas como uma ordem a um dos outros processos, exceto a primeira multiplicação, que ele mesmo executa. Cada processo, ao receber uma ordem de multiplicação, tem um comportamento semelhante, guardando a primeira multiplicação para si e distribuindo as outras para seus filhos na árvore de recursão.

Para montar uma árvore de recursão balanceada – ou seja, os processos não estarão concentrados em nenhuma das sub-árvores –, cada processo de *rank* i tem como filhos os 8 processos: i (o próprio processo), $i + 8^{nível}$, $i + 2*8^{nível}$, ..., $i + 7*8^{nível}$ (ver exemplo da figura). Se, num dado momento da recursão, não existir um processo com o rank que foi calculado, o próprio processo executa aquela multiplicação seqüencialmente, a partir daquele ponto.

Para efetuar o envio e recebimento do retorno das ordens de multiplicação, são utilizadas as primitivas **`MPI_Send(...)`** e **`MPI_Recv(...)`**, contendo a ordem e os elementos de cada matriz.

Testes

Para executar os algoritmos apresentados na seção anterior, foi utilizado o seguinte ambiente:

- **Hardware**
  - Intel(R) Core(TM) 2 Quad CPU Q6600 @ 2.40GHz
  - Bus: 1066 MHz
  - L2 Cache: 8 MB

    - L2 Cache Speed: 2.4 GHz
    - Manufacturing: 65 nm
    - 3 GB RAM

- **Software**
    - Ubuntu 9.04, kernel Linux 2.6.27-11-generic
    - gcc 4.3.3 (OpenMP 2.5)
    - gcc 4.4.0 (OpenMP 3.0)
    - LAM 7.1.2 / MPI 2

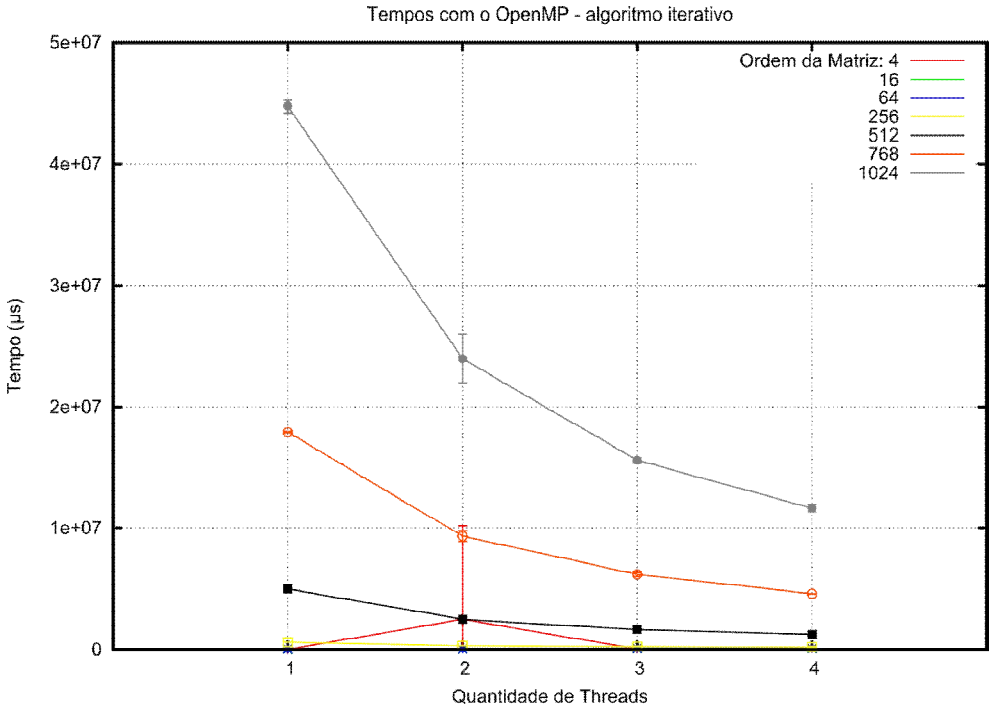
As simulações foram uma combinação de:

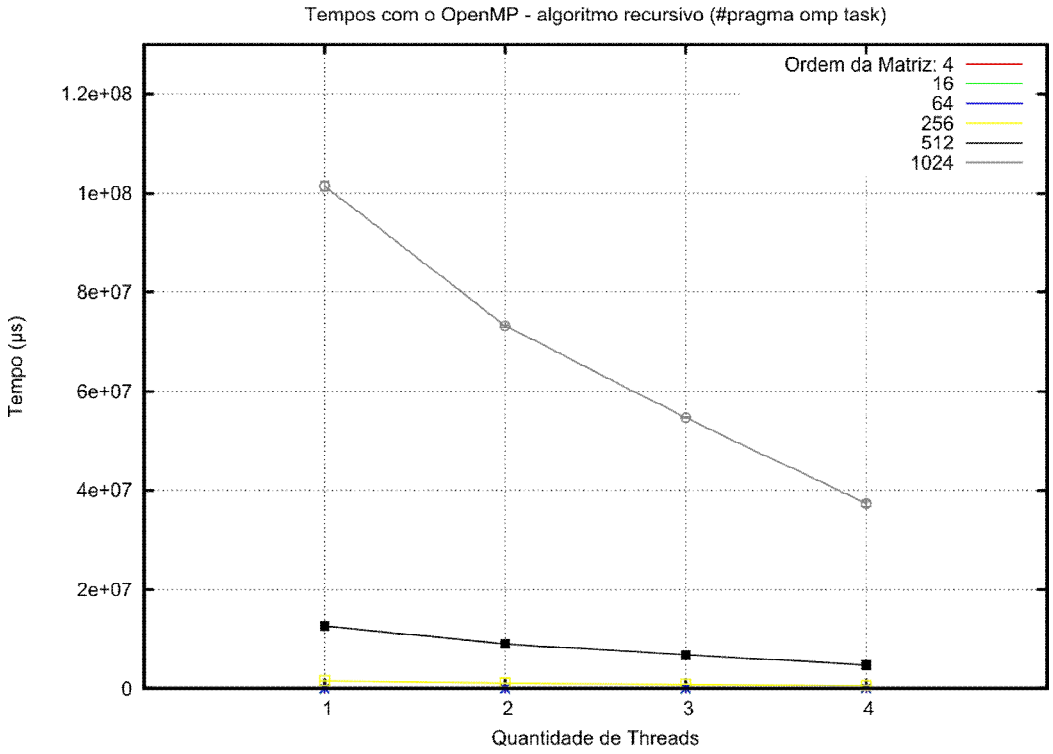
- **Número de processos, ou threads**: 1, 2, 3 e 4;
- **Implementação**: iterativo, recursivo com omp tasks e recursivo com MPI e
- **Ordem das matrizes**: 4, 16, 64, 256, 512 e 1024 (além de 768, no iterativo).

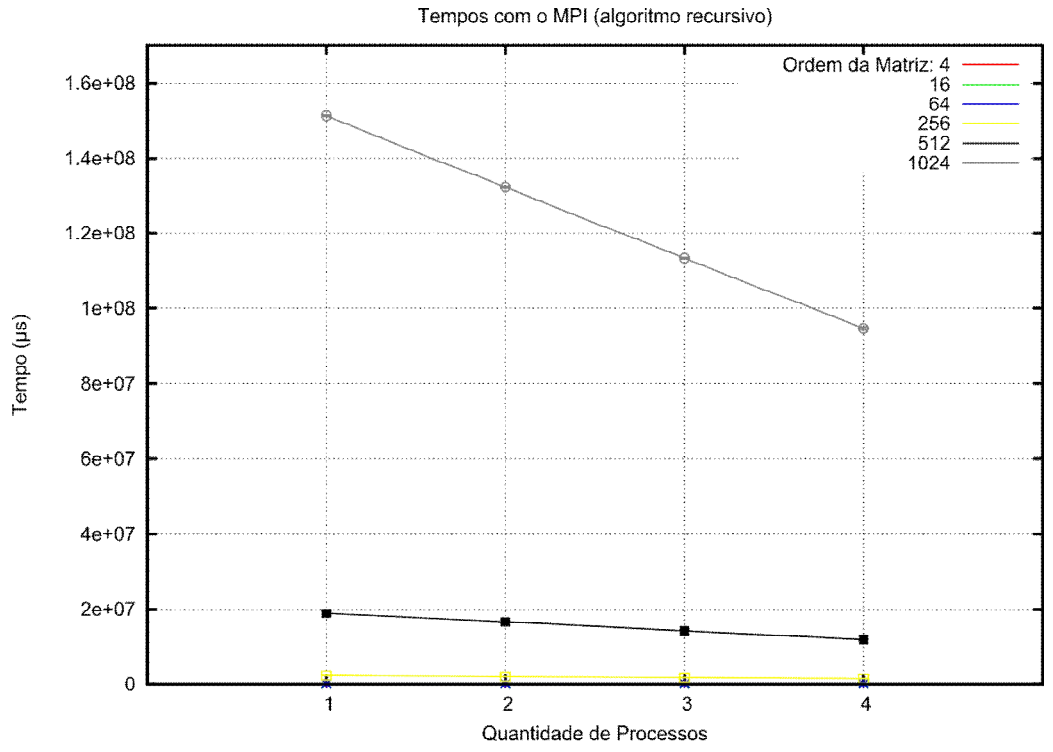
No total, foram pelo menos 72 combinações, cada uma sendo executada 10 vezes, para cálculo da média e do desvio-padrão.

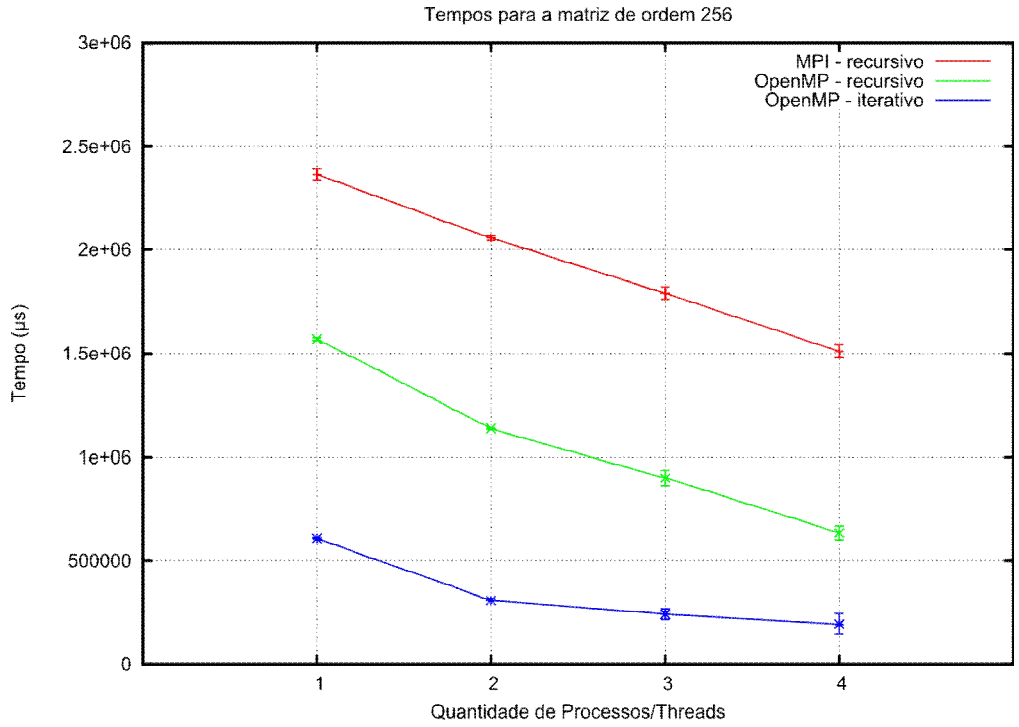
Obs.: Também se planejou fazer as simulações no SIMICS. Porém, devido a falhas no servidor de licenças, infelizmente isso não foi possível. Fica para um trabalho futuro.

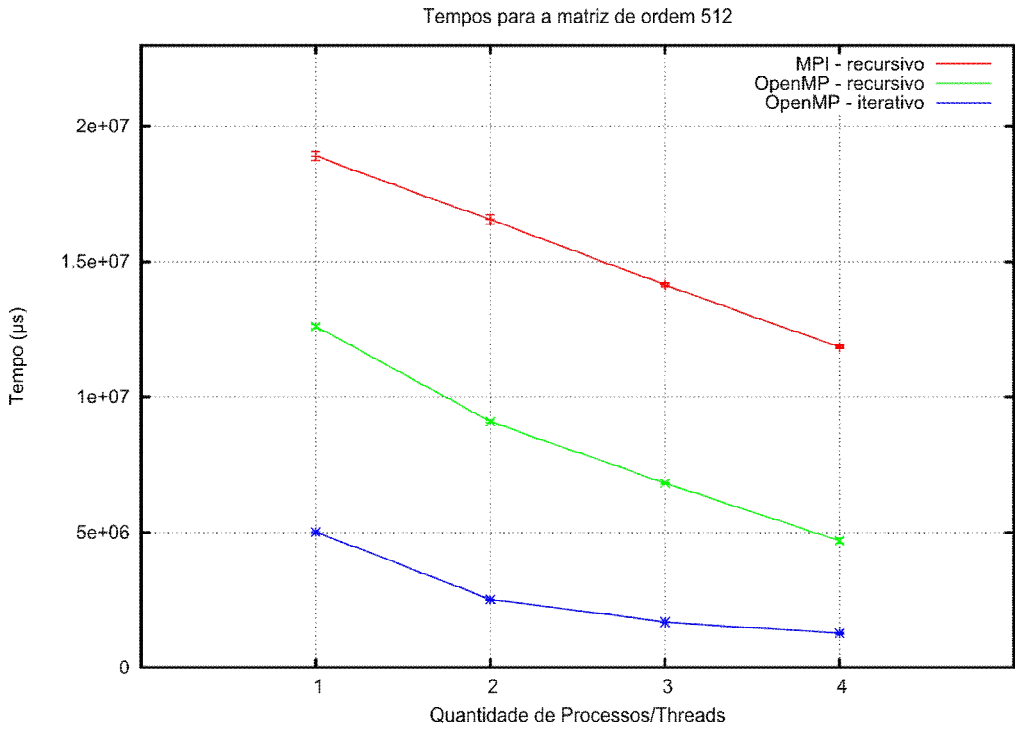
Resultados

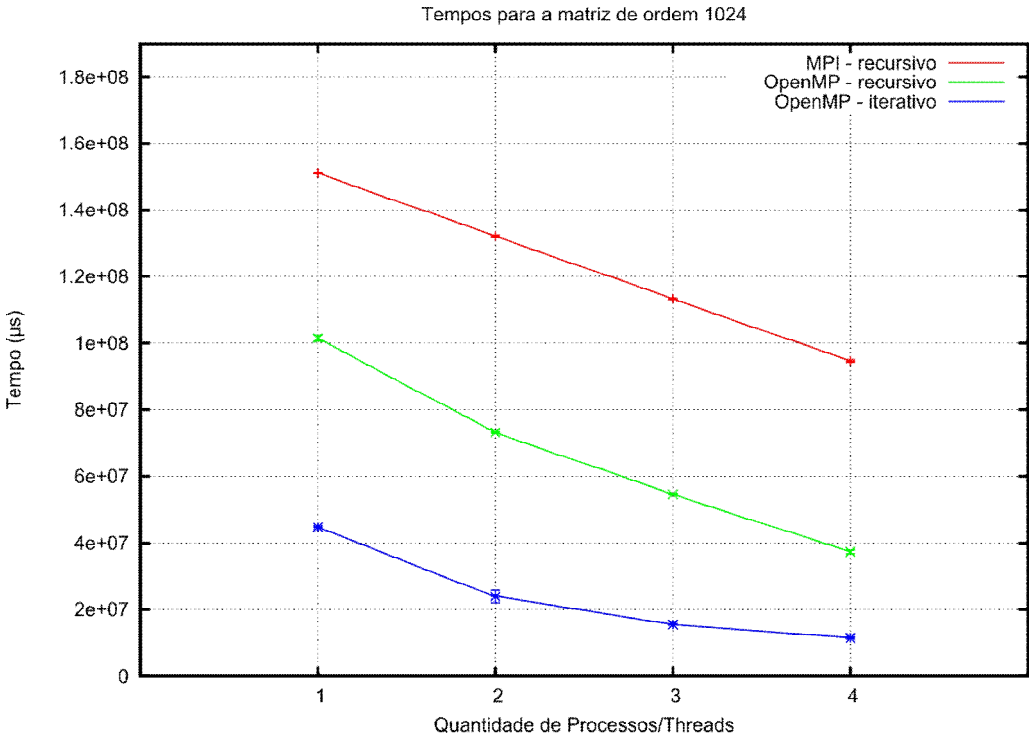


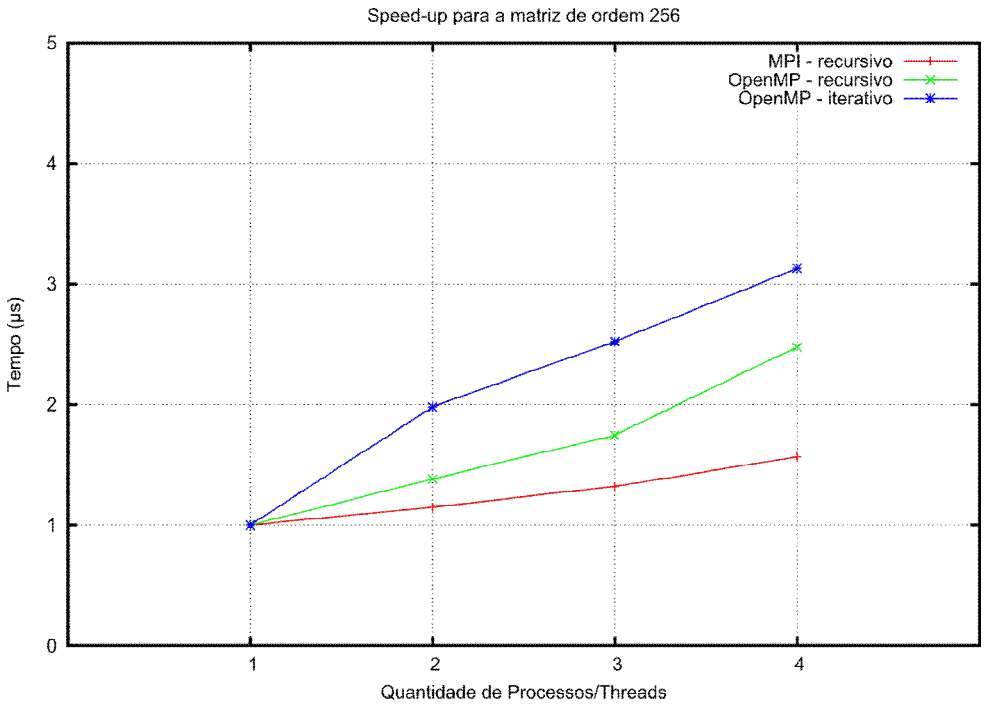

Tempos com o OpenMP - algoritmo recursivo (#pragma omp task)
Ordem da Matriz: 4
16
64
256
512
1024
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
Tempo (µs)
1
2
3
4
Quantidade de Threads

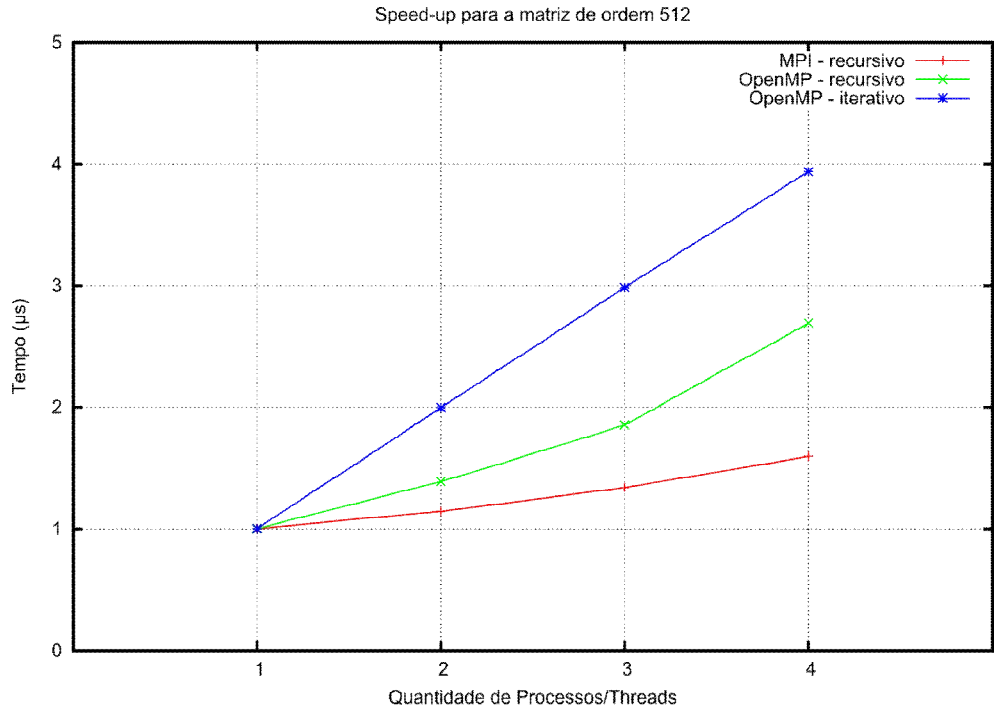

Tempos com o MPI (algoritmo recursivo)
Ordem da Matriz: 4
16
64
256
512
1024
1.6e+08
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
Tempo (µs)
1
2
3
4
Quantidade de Processos

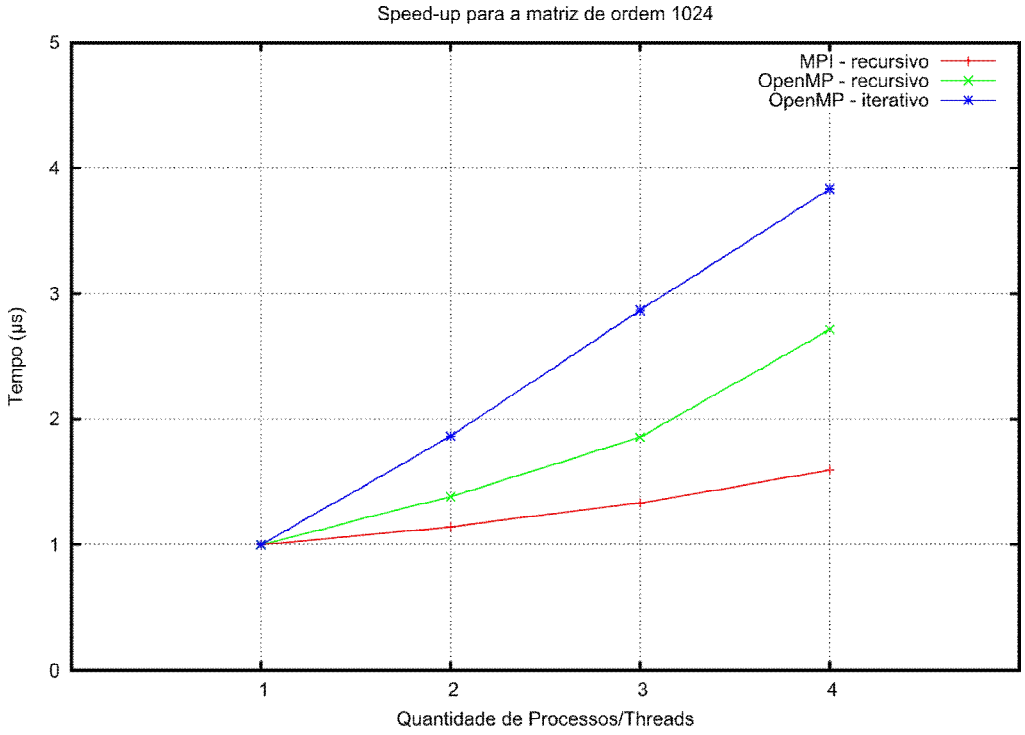

Tempos para a matriz de ordem 256
MPI - recursivo
OpenMP - recursivo
OpenMP - iterativo
Tempo (µs)
3e+06
2.5e+06
2e+06
1.5e+06
1e+06
500000
0
1
2
3
4
Quantidade de Processos/Threads


Tempos para a matriz de ordem 512
MPI - recursivo
OpenMP - recursivo
OpenMP - iterativo
Tempo (µs)
2e+07
1.5e+07
1e+07
5e+06
0
1
2
3
4
Quantidade de Processos/Threads


Tempos para a matriz de ordem 1024
MPI - recursivo
OpenMP - recursivo
OpenMP - iterativo
Tempo (µs)
1.8e+08
1.6e+08
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
1
2
3
4
Quantidade de Processos/Threads


Speed-up para a matriz de ordem 256
MPI - recursivo
OpenMP - recursivo
OpenMP - iterativo
Tempo (µs)
5
4
3
2
1
0
1
2
3
4
Quantidade de Processos/Threads


Speed-up para a matriz de ordem 512
MPI - recursivo
OpenMP - recursivo
OpenMP - iterativo
Tempo (µs)
Quantidade de Processos/Threads
0
1
2
3
4
5


Speed-up para a matriz de ordem 1024
MPI - recursivo
OpenMP - recursivo
OpenMP - iterativo
Tempo (µs)
Quantidade de Processos/Threads
0
1
2
3
4
5

Conclusões

A partir dos resultados encontrados, podemos tirar algumas conclusões, pelo menos para o tipo de aplicação que foi simulado. Por exemplo, o OpenMP, apesar de oferecer menor liberdade, teve uma performance significativamente melhor do que o MPI. O fato de usar memória compartilhada, ao invés de memória distribuída certamente foi determinante para isso. O uso do MPI é mais indicado, pois, quando em um ambiente de memória distribuída.

Outra questão é no que se refere a recursividade. Ficou evidente, com base nos resultados, que um algoritmo iterativo tem um desempenho consideravelmente melhor do que a sua versão recursiva. O sobrecusto causado pelo empilhamento de chamadas recursivas na memória do computador, assim como o gerenciamento das tasks (no OpenMP 3.0) e o envio e recebimento de sub-matrizes (no MPI) aumentaram de maneira significativa o tempo de cálculo do produto entre as matrizes.