Computação de Alto Desempenho

Relatório Trabalho Prático 1

Alunos:

Marco Simões, msimoes@student.dei.uc.pt

Nuno Lourenço, naml@student.dei.uc.pt

Coimbra, Abril de 2010

Índice

Introdução

No âmbito da disciplina de Computação de Alto Desempenho integrante do plano curricular do Mestrado em Engenharia Informática foi desenvolvido o seguinte trabalho com o objectivo de avaliar o desempenho de algoritmos paralelos para a avaliação de polinómios.

A estrutura do relatório apresenta inicialmente uma descrição do problema a analisar, seguida de uma descrição das implementações com a respectiva justificação das tecnologias usadas. Apresentamos os testes efectuados, os resultados obtidos e, finalmente, as conclusões extrapoladas.

Este trabalho permitiu-nos desenvolver as nossas capacidades de paralelismo de algoritmos, quer a nível do problema quer ao nível da implementação.

Descrição do problema

O problema passa por fazer a avaliação de polinómios de ordem variável para diferentes valores de teste. Assim, para um polinómio do tipo:

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

o programa recebe o grau do polinómio, uma lista de coeficientes [a_0 a_1 … a_n] e o valore de x a testar.

O output do programa é o valor da avaliação do polinómio para o valor de x fornecido.

Implementações

Nesta secção justificamos a tecnologia que o usámos para garantir o paralelismo e fazemos a descrição das diferentes implementações efectuadas.

Tecnologia de paralelismo utilizada

Para as implementações paralelas utilizámos OpenMp, uma API para multiprocessamento com recurso a memória partilhada compatível com C, C++ e Fortran. A escolha centrou-se na facilidade da utilização e integração do desenvolvimento em C (linguagem utilizada), pois baseia-se na simples inserção de directivas de compilação, rotinas e variáveis de ambiente no código fonte que causam o paralelismo no run-time da aplicação.

Implementações sequenciais

Aqui são descritas as diferentes implementações sequenciais efectuadas, salientando as suas principais características e complexidades. Todas elas têm apenas um fluxo de execução e não usam Openmp.

Basic

Dumb

A versão básica resolve o problema percorrendo a lista dos coeficientes e multiplicando-os pelo valor de teste elevado ao grau nesse coeficiente.

Algoritmo

```
result = 0

for( i = 0; i <= n; i++ )
      result += ( coef[i] * pow(test, i) )
```

Complexidade: Nesta versão, são efectuadas cerca de $\mathbf{(n^2 + n)/2}$ multiplicações e **n** adições.

Memória: A nível de memória apenas necessita de uma variável para ir mantendo o valor de result.

Optimized

A versão optimizada do algoritmo base faz uso de programação dinâmica. A ideia consiste em ao calcular x^3 usar o valor já calculado de x^2 em vez de começar do zero.

Algoritmo

```
result = 0
powerValue = 1

for ( i = 0; i <= n; i++ )
      result += (powerValue * coef[i])
      //calculate the next power value from the current
      powerValue *= test
```

Complexidade: Nesta versão, são efectuadas cerca de **2n** multiplicações e **n** adições.

Memória: Utiliza apenas duas variáveis auxiliares para ir mantendo o valor do resultado e do teste elevado ao expoente.

Horner

Este método faz decompões o polinómio da seguinte forma:

$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$

$= (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots)x + a_0$

Isto permite avaliar o polinómio numa só passagem, sem multiplicações extra para manter os power values, pois é sempre feita a multiplicação por x.

Algoritmo

```
result = coefs[n]

for ( i = n - 1; i >= 0; i-- )
      result = coefs[i] + test * result
```

Complexidade: Nesta versão, são efectuadas cerca de **n** multiplicações e **n** adições.

Memória: Utiliza apenas uma variável auxiliar para ir mantendo o valor do resultado.

Implementações paralelas

Aqui são descritas as implementações paralelas que efectuámos. As partes paralelas e as sequenciais são explicitamente identificadas, e o número de núcleos de processamento necessários é sempre referido.

Prefix sum

Este algoritmo é uma variante do *prefix sum problem*, e é utilizado para calcular os power values em log(n) passos. A figura seguinte descreve graficamente a forma como o algoritmo actua para cálculo dos valores.

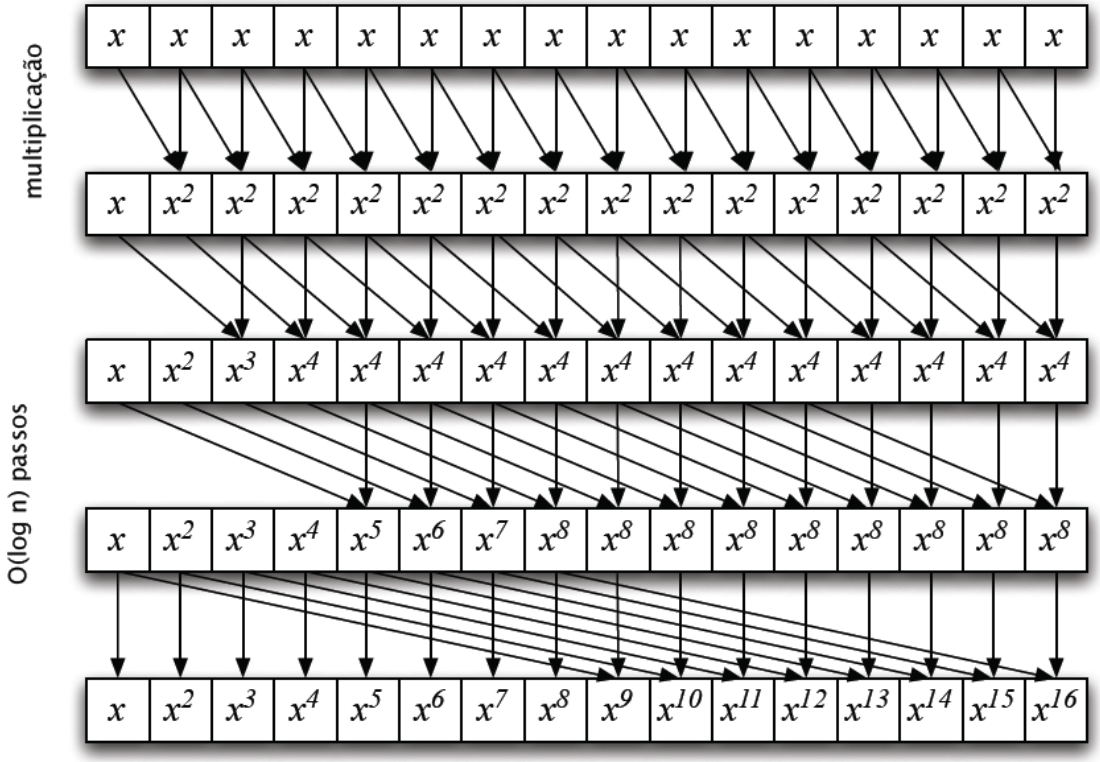


Figura 1 - Variante do prefix sum

n processors

Esta implementação aloca uma thread para cada grau do polinómio. Na figura acima, podemos visualizar como associando uma thread por coluna. Em cada iteração, as threads têm que multiplicar o seu valor ao valor de uma casa a um offset (1, 2, 4, 8, … n) de distância.

Algoritmo

```
#parallel com reduction(+:result)
{

// tid is the id of the thread, from 0 to n

// initialize values with test value
values[tid] = test

for ( i = 0; i < log2(n); i++ )
      aux = values[tid]

      #sync threads here

      if ( tid + offset < n )
            values[tid + offset] *= aux

      #sync threads here

      #thread principal
            offset *= 2


// finally, multiply with its coeficient
result = values[tid] * coef[tid + 1]

}
```

Complexidade: Nesta versão, são efectuadas cerca de $\log_2$**n** multiplicações e **n** adições (adições efectuadas por reduction).

Memória: Utiliza uma variável auxiliar para ir mantendo o valor do resultado, mais uma variável auxiliar para cada thread => **n + 1**.

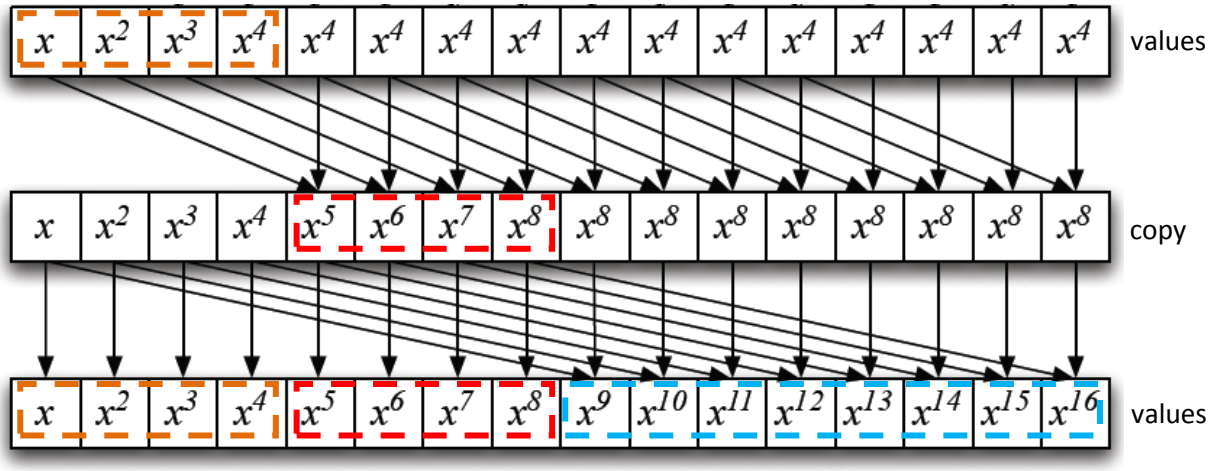
Threads: Utiliza n threads

Pontos de Sincronização: 2 por iteração => 2 $\log_2$n

k processors

Esta implementação é uma versão do algoritmo anterior optimizada para o uso de apenas k threads em vez de uma thread para cada grau do polinómio. Desta forma, ganhamos os tempos de context switch, criação e destruição das threads. Nesta versão, cada thread executa mais operações que na versão anterior, mas como a arquitectura onde o algoritmo é corrido não consegue executar n threads em simultâneo esta implementação corre mais depressa.

A imagem abaixo demonstra para uma iteração (como no algoritmo anterior, há $\log_2 n$ iterações) que operações são precisas fazer.



Valores mantêm-se duas iterações atrás (não é necessário copiar)

Valores iguais à iteração anterior (necessário copiar da iteração anterior)

Novos valores

Figura 2 - prefix sum v2: operações necessárias numa iteração

Em cada iteração são efectuados os passos a vermelho e a azul em paralelo pelas k threads, pois não tem dependências entre os elementos do array. A utilização de dois arrays (values e copy) permite não haver necessidade de sincronização de threads na execução dos passos a azul, pois os dados do novo array dependem apenas do array anterior. A secção a laranja mantêm-se, não precisa ser actualizada. O algoritmo abaixo demonstra claramente os passos bem identificados em cada iteração.

Algoritmo

```
offset = 1
for ( it = 0; it < log2(n); it++ )

      #parallel for // for dividido pelas k threads (vermelho)
      for (i = offset/2; i < offset; i++)
            copy[i] = values[i]

      #parallel for // for dividido pelas k threads (azul)
      for (i = 0; i < degree-offset; i++)
            copy[i + offset] = values[i] * values[i+offset]

      // actualiza o offset
      offset *= 2

      // troca arrays copy e values
      swap( copy, values )

// multiplicar coeficientes por power values e somar para result
#parallel for com reduction(+:result)
for(i = 0; i < degree; i++)
      result += values[i] * coef[i + 1]

// somar coeficiente sem power value
result += coef[0]
```

Complexidade: Esta implementação faz cerca de $\frac{\left(\sum_{i=0}^{\log(n)-1} n-\frac{n}{2^i}\right)+n}{k}$ multiplicações por thread e cerca de n/k adições por thread

Memória: A nível de memória, utiliza de 2 arrays de tamanho n, o que se revela um problema na avaliação de polinómios de grau muito grande

Threads: Utiliza k threads (dadas as características das nossas máquinas, definimos o número de threads como 2).

Estrin

Este algoritmo é uma solução paralela que permite realizar as operações de adição e multiplicação de forma paralela, para que, durante o cálculo do polinómio, possamos usufruir das novas arquitecturas multiprocessador para realizar os cálculos.

O seu funcionamento é simples: Dado o polinómio $P_n(x)= C_0 +C_1x +C_2x^2 +C_3x^3 ... +C_nx^n$ podemos rearranjar o polinómio em sub-expressões do tipo (A + Bx) e do tipo x^{2n}. Assim depois de rearranjado, a forma final do polinómio P é: $P_n(x)= (C_0 +C_1x) + (C_2 +C_3x)x^2 + ((C_4 +C_5x) + (C_6 +C_7x) x^2))x^4$.

Algoritmo

```
length = degree + 1

for( it = 0; it < log2(n); it++ )

      #parallel for
      for ( i = 0; i < length / 2; i++ )
            copy[ i ] = values[ i * 2 ] + values[ i * 2 + 1 ] * test

      //copy last value if length is odd
      if ( length % 2 == 1 )
            copy[ length / 2 ] = values[ length - 1 ]


      test *= test
      length = ceil( length / 2.0 )

      swap( copy, values )

result = values[0]
```

Complexidade: Neste algoritmo, cada thread executa cerca de $\frac{\left(\sum_{i=1}^{\log(n)} \frac{n}{2^i}\right)}{k} + \log(n)$ multiplicações e um igual número de somas.

Memória: A nível de memória utiliza de 2 arrays de tamanho n.

Threads: Utiliza k threads (usamos duas por melhor ajuste à nossa arquitectura). Num modelo semelhante ao do prefixSum, este algoritmo permite uma implementação com n threads, que vai dividindo por 2 a cada iteração. No entanto, depois de alguns testes preliminares no algoritmo prefixSum com n threads, decidimos não fazer essa implementação. Mais à frente demonstramos nos testes efectuados a razão de não optarmos por tal implementação.

Horner parallel

Este algoritmo pretende ser uma optimização do da versão Horner sequencial, por forma a podermos aproveitar todas as funcionalidades das arquitecturas multiprocessador. Para isso dividimos as operações por k processadores, que trabalham de forma independente.

Quando usado com duas threads, uma boa visualização passa por imaginar uma thread a percorrer os coeficientes pares e outra a percorrer os coeficientes impar.

Algoritmo

```
// calcular x^k
testK = pow( test, k )

#parallel
      //tid é o id da thread de 0 a k

      aux = coef[ n - k - tid ] + testK * coef[ n - tid ]

      for ( it = 1; it < (n - k + 1) / k; it++ )
            aux = coef[ n – (it - 1) * k - tid ] + testK * aux


      if (tid < (aux - k + 1) % k)
            aux = coef[ tid ] + testK * aux;

      resultSpace[tid] = aux;

// juntar os resultados de todas as thread utilizando o método
de horner sequencial
for (i = k - 2; i >= 0; i--)
      resultSpace[i] = test * resultSpace[i+1] + resultSpace[i]

result = resultSpace[0]
```

Complexidade: Cada thread faz cerca de $\frac{n}{k}$ somas e multiplicações. Para juntar tudo no final temos mais 2*k - 2 multiplicações e k-1 somas.

Memória: A nível de memória, temos apenas uma variável auxiliar para cada thread (k variáveis), uma variável para manter x^k e um array de tamanho k.

Threads: Utiliza k threads, que no nosso caso optámos por usar duas, devido a características da plataforma.

Testes

Para testar a performance dos nossos algoritmos decidimos montar uma experiência onde variamos os valores do grau do polinómio, desde valores muito pequenos (e.g. 10), até valores muito grandes (e.g. 14676620). Os valores do grau dos polinómios são incrementados tendo em conta sempre os dois valores anteriores (semelhante a sequencia de fibonacci). Isto é, se começarmos com os valores 40 e 50, o grau do nosso polinómio irá crescer da seguinte forma:

40, 50, 90, 140, 230, 370, 600, 970, 1570

Para o número de avaliações decidimos escolher um número alto (100 valores de teste), para ver como é que o algoritmo se comportava. Cada input é avaliado 3 vezes, e no fim fazemos a média dos resultados obtidos, para que assim os dados sejam estatisticamente mais relevantes (despistando variações mais pontuais de performance).

Fizemos ainda algumas divisões relativamente ao grau, pois alguns algoritmos são demasiado pesados e não conseguiam correr em tempo útil nas máquinas que tínhamos disponíveis.

A divisão está feita na seguinte tabela:

Experiência	Grau dos Polinómios	Algoritmos
1	10, 40, 50	Todos os Algoritmos
2	90, 140, 230, 370, 600, 970, 1570, 2540, 4110, 6650, 10760, 17410, 28170, 45580, 73750	Todos os algoritmos, com excepção do PrefixSumV1
3	119330, 193080, 312410, 505490, 817900, 1323390, 2141290, 3464680, 5605970, 9070650, 14676620	Apenas os algoritmos: Estrin, Horner Parallel, Horner, Basic Optimized

Tabela 1 - Características dos testes: graus dos polinómios usados em cada algoritmo

Os testes foram efectuados num computador portátil Macintosh (APPLE MACBOOK Pro) com um Intel CORE 2 Duo 2,26 GHz e 2 GB de memória RAM, sobre o sistema operativo MAC OSX version 10.5.8.

Resultados

Tendo em conta os testes que foram especificados acima, e depois de realizadas as experiências, os resultados obtidos foram os seguintes:

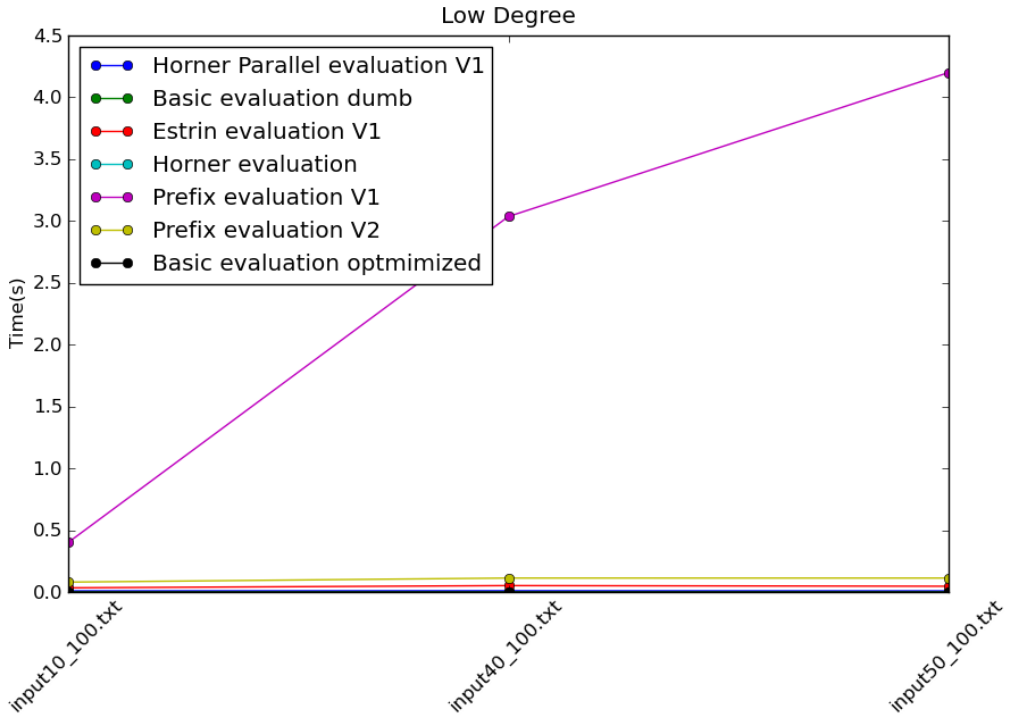


Figura 3 - Gráfico da experiência 1

Como podemos observar pelo gráfico acima, todos os algoritmos conseguem resolver o problema num tempo muito próximo de 0 segundos, excepto o PrefixSumV1 que demora bastante tempo e cresce muito rapidamente em função do tamanho do input. Isto deve-se ao facto de o algoritmo utilizar n (grau do polinómio) threads para resolver o problema. As operações de criação, *context-switch* e eliminação de threads levam a uma degradação da performance quando comparada com a dos outros algoritmos, que utilizam um número controlado de threads ou são apenas sequênciais.

Estes resultados permitiram-nos ainda concluir que para valores maiores de n, o algoritmo não iria conseguir produzir resultados em tempo útil na nossa arquitectura computacional de testes. Isto deve-se principalmente à proporcionalidade directa entre o número de threads e o grau do polinómio, e ao facto das nossas arquitecturas apenas conseguirem correr duas threads em simultâneo.

Esta implementação foi assim retirada dos testes seguintes devido à sua discrepância de performance visível nesta primeira experiência.

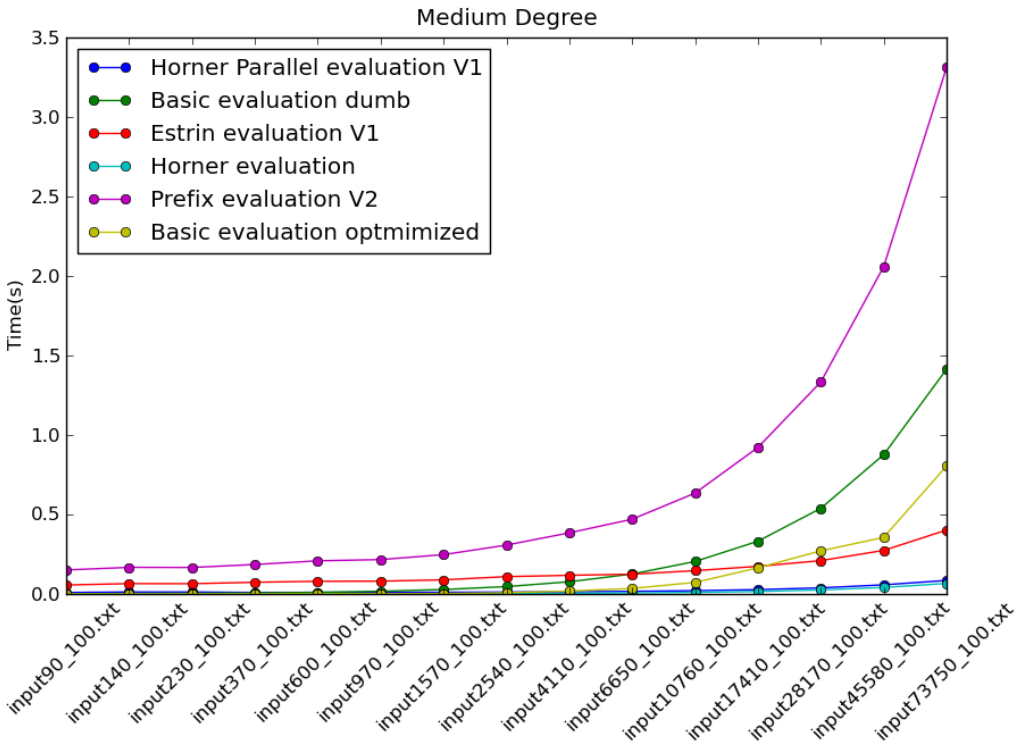


Figura 4 - Gráfico da experiência 2

Neste gráfico podemos observar que comparado com todas as outras implementações, o algoritmo PrefixSumV2 é o que tem pior performance, mesmo comparado com a versão basic sem optimizações nenhumas. A complexidade deste algoritmo, como tínhamos observado acima, é a mais elevada, tendo em conta que só conseguimos utilizar simultaneidade de 2 threads.

Observamos que a versão Basic sem optimizações segue a versão do prefix de forma próxima, e de seguida a versão optimizada do mesmo algoritmo.

É interessante verificar que a implementação do Estrin, quando o n começa a crescer mais significativamente, ganha aos algoritmos básicos (ambos).

Uma disputa renhida verifica-se entre as duas versões de Horner, a sequencial e paralela.

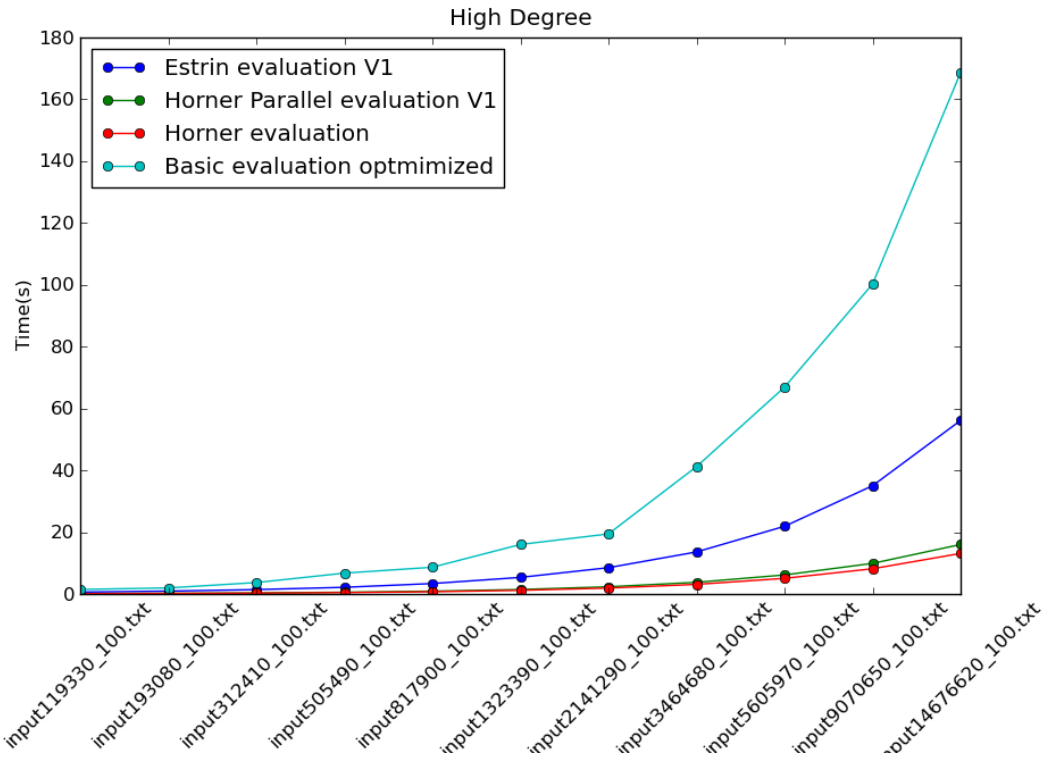


Figura 5 - Gráfico da experiência 3

Esta última experiência utiliza polinómios de valor elevado de grau. Podemos verificar que as conclusões verificadas na experiência 2. A versão paralela do Horner perde ligeiramente para a versão sequencial, e o Estrin ganha significativamente à versão básica optimizada.

Conclusões

Podemos concluir deste trabalho que conseguimos arranjar algoritmos paralelos que ganham em complexidade às versões sequenciais. No entanto, o hardware para execução não nos permite grande paralelismo. No nosso caso, apenas temos 2 núcleos de execução, e nada nos garante uma execução simultânea de duas threads. O sistema operativo pode fazer o Schedule das duas threads no mesmo núcleo, enquanto diferentes programas executam no outro. Desta forma, os algoritmos sequenciais têm sérias vantagens em relação aos paralelos.

A simplicidade do problema também não permite verificar as vantagens principais do paralelismo. Se houvessem chamadas ao sistema, acessos a disco ou operações mais bloqueantes poderíamos observar melhor as vantagens do paralelismo.

A implementação paralela do Horner mostrou, mesmo contra todas as dificuldades da arquitectura, excelentes resultados. Fica bastante próximo do resultado da versão sequencial, tendo menor complexidade e necessidade de pouca memória.

Este trabalho permitiu-nos desenvolver competências no desenvolvimento de aplicações paralelas, desde o nível algorítmico ao nível de implementação.