Experimentos com linguagens, paralelismo e alto consumo de CPU

Marcelo Rabello Rossi e Alfredo Goldman
Instituto de Matemática e Estatística, Universidade de São Paulo

Introdução

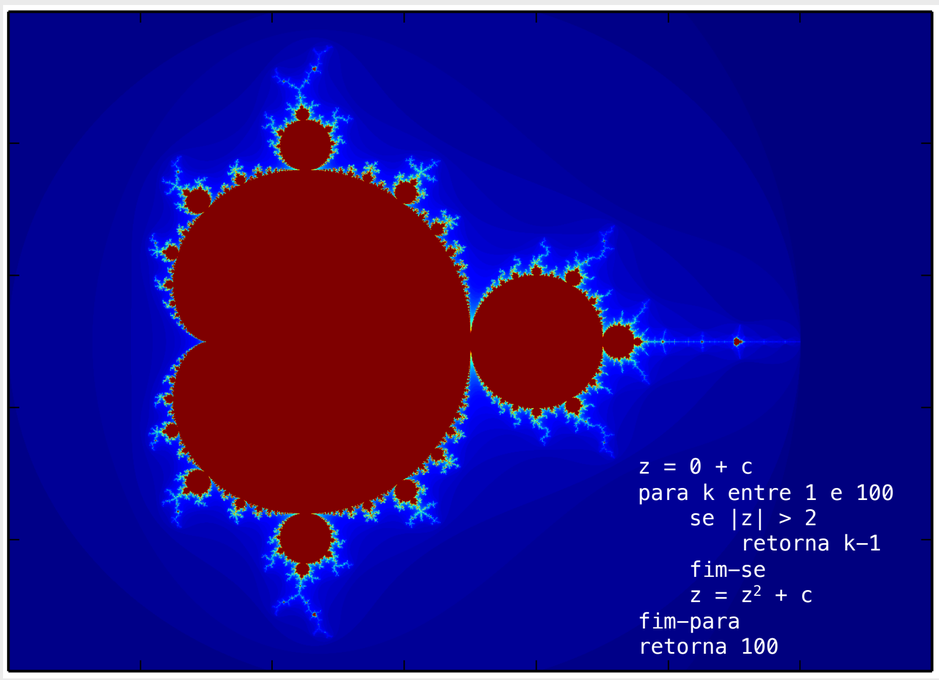
- A partir de 2004 os processadores atingem um patamar em termos de frequência e consumo de energia
- Eles passam a evoluir de outras formas, sendo uma delas o aumento do número de núcleos de processamento
- Para se obter os benefícios dessa evolução, passa a ser necessário escrever softwares concorrentes e paralelos
- Se existem vários sistemas e linguagens que permitem o paralelismo, passa a ser interessante estudar a usabilidade e o desempenho de cada um para um determinado propósito de interesse

Objetivos

- Estudar o comportamento de quatro liguagens – **Go, Julia, Python e C/OpenMP** – em um contexto perfeitamente paralelo de uso intenso de CPU
- Como ferramenta de comparação, utilizou-se o algoritmo de obtenção do **conjunto de Mandelbrot**
- As métricas observadas foram: desempenho, *speedup* em latência e tamanho do código
- As variáveis estudadas foram: tipo de hardware (local ou *cloud based*), número de processadores, sistema operacional, paradigma de programação e tipo de escalonamento de processos

Conjunto de Mandelbrot

Conjunto dos números complexos c para os quais a função $z = z^2 + c$ não diverge quando iterada a partir de $c = 0$



Julia

```julia
function mandelbrotorbit(c)
    z = Complex128(0,0) + c
    for k = 1:100
        if abs(z) > 2
            return k-1
        end
        z = z^2 + c
    end
    return 100
end

function mandelbrot(x::Float64)
    Z = [complex(x,y) for y in Y]
    return map(mandelbrotorbit, Z)
end

X = linspace(-2.5, 1.0, rows)
Y = linspace(-1.25, 1.25, columns)

N = pmap(mandelbrot, X)
```

C

```c
int mandelbrotorbit(Complex c) {
    Complex z;
    z.re = 0.0; z.im = 0.0;
    z = addComplex(multComplex(z, z), c);
    for (int i = 1; i <= 100; i++) {
        if (absComplex(z) > 2) {
            return i - 1;
        }
        z = addComplex(multComplex(z, z), c);
    }
    return 100;
}

int** mandelbrot(Complex** inputmat) {
    int **outputmat = malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; i++){
        outputmat[i] = malloc(columns * sizeof(int));
    }

    #pragma omp parallel for schedule(dynamic, chunk_size) num_threads(nthreads)
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < columns; j++) {
            outputmat[i][j] = mandelbrotorbit(inputmat[i][j]);
        }
    }
    return outputmat;
}
```

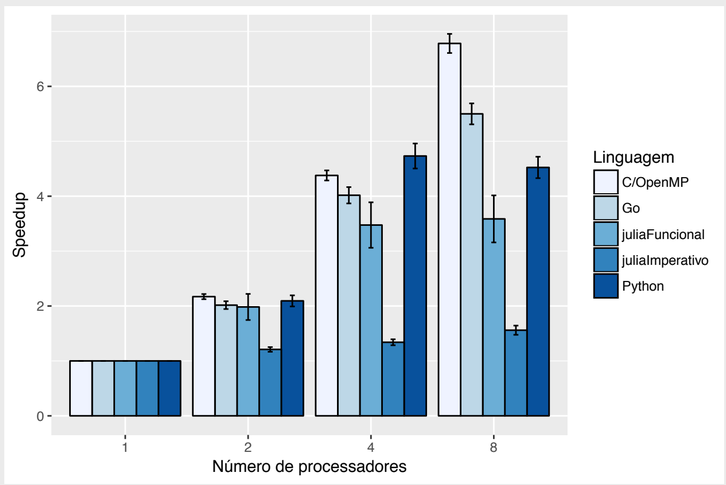
Resultados

- Os experimentos tiveram variabilidade média 8x maior na GCE
- Julia (funcional) apresentou o menor tempo de execução
- C/OpenMP apresentou o maior ganho em desempenho com a paralelização
- Dependendo da linguagem e do tamanho do problema, não vale a pena paralelizar

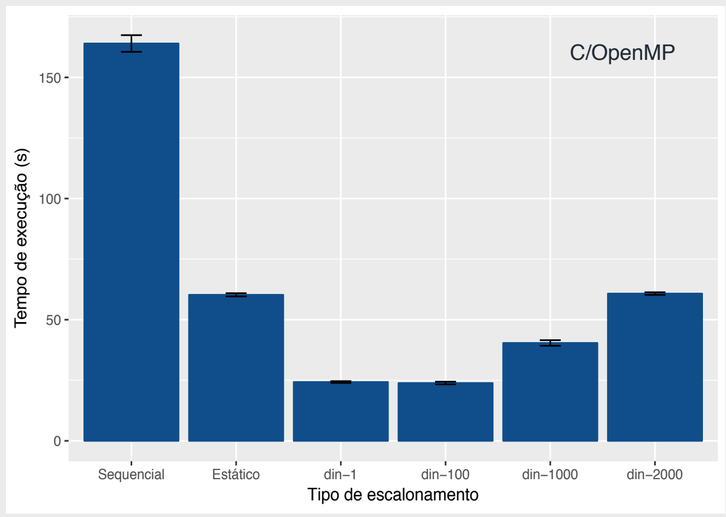
	Go					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
[mbp]	25,02 ± 0,06	99,39 ± 0,14	396,19 ± 0,16	5,42 ± 0,09	21,32 ± 0,26	83,36 ± 0,43
[gce]	32,0 ± 3,1	15,6 ± 4,5	460 ± 16	5,26 ± 0,07	21,18 ± 0,06	83,73 ± 0,43
	JuliaFuncional					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
[mbp]	2,78 ± 0,03	7,91 ± 0,05	28,95 ± 0,96	3,47 ± 0,03	5,04 ± 0,04	9,86 ± 0,09
[gce]	3,37 ± 0,32	9,47 ± 0,81	30,2 ± 3,6	3,99 ± 0,07	5,33 ± 0,10	8,41 ± 0,13
	JuliaImperativo					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
[mbp]	1,94 ± 0,09	7,50 ± 0,11	26,9 ± 1,3	5,68 ± 0,04	7,46 ± 0,05	14,79 ± 0,35
[gce]	2,03 ± 0,26	6,83 ± 0,62	24,25 ± 0,42	6,57 ± 0,05	8,17 ± 0,08	15,55 ± 0,79
	Python					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
[mbp]	41,25 ± 0,64	164,8 ± 3,1	657 ± 11	12,19 ± 0,52	47,24 ± 0,81	186,5 ± 3,1
[gce]	48,9 ± 1,1	194,1 ± 4,8	798 ± 30	11,61 ± 0,33	44,9 ± 0,83	176,3 ± 3,5
	C/OpenMP					
	Sequencial			Paralelo		
	3500	7000	14000	3500	7000	14000
[mbp]	3,26 ± 0,03	13,10 ± 0,30	52,6 ± 1,5	0,78 ± 0,01	3,23 ± 0,03	12,97 ± 0,06
[gce]	9,90 ± 0,19	40,54 ± 0,41	164,7 ± 3,4	1,580 ± 0,022	6,15 ± 0,06	24,28 ± 0,36

Tabela 1. Tempos de execução do algoritmo de obtenção do conjunto de Mandelbrot, em segundos, corrigidos pela subtração dos tempos de geração das matrizes de entrada. As variáveis consideradas foram: sistema utilizado ([mbp]: MacBook Pro Core i7 Quad Core 16GB RAM, [gce]: Google Compute Engine 8 vCPUs 32 GB RAM) tipo de execução, tamanho da entrada e linguagem de programação utilizada na implementação do algoritmo

C/OpenMP e Go apresentaram os maiores *speedups* em latência



Devido aos diferentes workloads de cada tarefa durante a obtenção do conjunto de Mandelbrot, o escalonamento dinâmico com *chunk sizes* pequenos melhora o desempenho do algoritmo



A linguagem que mais foi impactada pelo paralelismo em termos de tamanho do código foi Go, que ganhou 10 linhas no total. Duas de importação de bibliotecas, uma de definição de variável, e as outras 7 de instruções de paralelismo.

Conclusões

- Para problemas desse tipo, pequenas adições ao código trazem grandes ganhos em desempenho
- O maior ganho em desempenho ocorreu em C/OpenMP, com escalonamento dinâmico de tarefas e 8 núcleos de processamento: a versão paralela executou ~7x mais rápido do que a versão sequencial
- Os programas implementados em Julia utilizando paradigma funcional mostraram alto desempenho mesmo em modo sequencial, obtendo os melhores tempos de execução. Com a paralelização, adicionando apenas 3 linhas, houve um ganho de ~4x em tempo de execução.
- O desempenho comparável a C, a simplicidade do código e a existência de diversas bibliotecas otimizadas fazem de Julia a melhor escolha geral entre as linguagens estudadas.
- Go pode ser a melhor opção em situações nas quais são necessárias uma grande quantidade de threads, devido à leveza de suas *goroutines*.
- As implementações de paralelismo das linguagens mais atuais estão, a cada dia mais, tornando o paralelismo acessível mesmo aos programadores mais leigos.