MC458 - Laboratório 1: Algoritmos de Ordenação

September 28, 2013

Conteúdo

Capítulo 1

Métodos

1.1 Improved cache memory use

One of the greatest optimizations we achieved, was the better use of the cache memory. Some data intensive reading algorithms can benefit from the cache memory, by accessing sequential memory addresses. With lesser cache misses, the algorithm has no need to search the data in the RAM memory, which is much slower than cache memories.

Capítulo 2

Otimizações

2.1 Sgemm

```
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b;
    }
    C[mm+nn*ldc] = C[mm+nn*ldc] * beta
        + alpha * c;
  }
}
```

Listing 2.1: Parboil sequential source code for sgemm

```
#pragma omp parallel for collapse (2)
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b;
    }
    C[mm+nn*ldc] = C[mm+nn*ldc] * beta
        + alpha * c;
  }
}
```

Listing 2.2: Parboil parallel source code for sgemm

For the matrices multiplication algorithm, the matrices are represented as arrays, so, we improved the memory access by transposing the matrices, this way, in the inner loop, the index i is only added instead of multiplied to calculate the line and column position of the matrix in the array. We needed then to aloccate auxiliary arrays to represent the transposed matrices and then transpose while making the copy from one array to another.

To avoid the increase of the execution time of the algorithm, we parallelized this transpositioning. This can be done because it has no loop-carried dependencies and the data is distributed almost equally for all threads without concurrency, in other words, it is high parallelizable.

We created the variables aux1 and aux2 because in the inner loop, these values are constant and in serial and parboil parallel versions, these constant is calculated every iteration of the inner loop wasting some time in unnecessary multiplications. The access to the array is almost sequential after this changes. using better the memory, caused some great improvement in execution time of this algoritm.

```
tb = (float *) malloc(ldb * n * sizeof(float));
# pragma omp parallel for collapse(2)
for (i = 0; i < ldb; i++)
  for (j = 0; j < n; j++)
    tb[j * n + i] = B[i * n + j];

ta = (float *) malloc(lda * m * sizeof(float));
# pragma omp parallel for collapse(2)
for (i = 0; i < lda; i++)
  for (j = 0; j < m; j++)
    ta[j * m + i] = A[i * m + j];

# pragma omp parallel for private(nn, c, i, a, b, mm) collapse(2) schedule (static)
for (mm = 0; mm < m; ++mm) {
  for (nn = 0; nn < n; ++nn) {
    c = 0.0f;
    aux1 = mm * lda;
    aux2 = nn * ldb;
    for (i = 0; i < k; ++i) {
      c += ta[i + aux1] * tb[i + aux2];
    }
    C[mm+nn*ldc] = C[mm+nn*ldc] * beta + alpha * c;
  }
}
```

Listing 2.3: Our parallel source code for sgemm

Capítulo 3

Análise e Resultados

- Vetor Ordenado
- Vetor Invertido
- Vetor Constante
- Vetor Aleatório

3.1 Algoritmos $O(n^2)$

3.1.1 Bubble Sort

3.2 Comparações dos Algoritmos

Tabela 3.1: Tabela dos tempos para os vetores aleatórios

N	Bubble Sort (us)	Insertion Sort (us)	Merge Sort (us)	Heap Sort (us)	Quick Sort (us)
20	0.54	0.38	2.18	0.89	0.33
40	2.06	1.28	4.74	2.09	0.85
60	5.41	4.38	2.64	3.38	1.25
80	5.19	6.54	3.71	4.66	2.21
100	6.30	5.42	4.69	4.15	0.97
120	9.86	5.62	5.61	2.64	1.22
140	13.29	8.16	7.16	3.21	1.62
160	18.98	10.39	7.75	3.97	1.71
180	24.19	13.20	8.81	4.58	1.87
200	31.50	17.26	9.77	5.07	2.18
220	39.08	18.71	10.73	5.64	2.30
240	46.73	24.89	11.97	6.38	2.54
260	55.65	26.57	13.28	6.89	3.06
280	62.90	32.87	14.36	7.59	3.59
300	71.83	36.10	15.41	8.38	3.74
320	82.09	38.36	16.36	8.99	3.78
340	89.37	48.51	17.87	9.87	4.25
360	103.90	52.94	19.01	10.53	4.32
380	114.72	57.14	20.40	11.57	5.18
400	127.54	65.93	21.87	12.13	4.84
420	138.09	77.04	23.34	13.28	5.16
440	150.81	77.11	25.22	13.84	5.84
460	163.94	85.78	27.67	14.32	6.30
480	177.77	95.85	28.10	15.50	7.15
500	192.97	97.70	31.44	15.79	6.24
520	209.49	108.34	32.95	17.27	6.62
540	219.24	119.94	35.29	18.61	7.30
560	237.04	125.29	36.40	18.91	7.54
580	247.24	140.73	38.26	20.31	8.09
600	270.45	147.62	40.02	20.88	8.90
620	284.33	149.17	42.12	21.62	9.62
640	309.54	166.93	43.10	22.76	9.35
660	315.12	178.41	45.89	23.40	10.78
680	342.33	188.27	48.52	25.07	11.13
700	358.19	197.32	49.73	26.47	11.25
720	373.96	215.76	51.21	26.79	12.10
740	397.99	222.46	53.13	27.65	14.54
760	411.69	232.85	54.71	29.08	13.95
780	439.17	245.02	58.36	29.81	15.68
800	461.42	264.66	59.94	31.70	15.90
820	476.24	276.48	61.90	32.29	17.69
840	496.48	285.15	64.12	34.03	18.62
860	515.91	292.42	65.49	33.94	20.76
880	539.34	309.45	68.73	35.85	22.05
900	559.42	322.18	69.30	35.84	23.26
920	592.42	332.22	72.63	37.46	24.07
940	616.82	353.43	74.38	38.75	25.46
960	645.81	372.20	76.63	40.61	27.16
980	668.59	385.35	80.79	40.99	27.79
1000	687.16	403.35	82.07	41.39	30.73

Capítulo 4

Referências Bibliográficas

- http://en.wikipedia.org/wiki/Bubble_sort
- http://en.wikipedia.org/wiki/Insertion_sort
- http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion_sort.html
- http://en.wikipedia.org/wiki/Merge_sort
- http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge_sort.html
- http://en.wikipedia.org/wiki/Heapsort
- http://pages.cs.wisc.edu/~paton/readings/Old/fall01/HEAP-SORT.htm
- http://en.wikipedia.org/wiki/Quicksort
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/quickSort.htm