September 28, 2013

Conteúdo

Capítulo 1

Methods

1.1 Improved Memory Use

One of the greatest optimizations we applied in several algorithms was the better use of the cache memory and the main memory. Some data intensive reading algorithms can benefit from the cache memory by accessing sequential adresses from the main memory, with lesser cache misses, the algorithm has no need to search the data in the main memory, which is much slower than low level memories.

By changing the way the memory is accessed, the performance was improved in the sequential version of some algorithms, improving even the performance over some simple parallelized versions.

Several algorithms in this benchmark set have nested loops and access some structure, like a array or a matrix, inside the inner loop. Also, some algorithms have inefficient index choice for the inner loop, turning the memory access into a slow and repetitious task. Some changes that improved the performance include the remapping of the structure or/and the better choice of the order for the indexes in the nested loops.

Capítulo 2

Optimizations

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, it is used an array-represented structure, so, we improved the memory access by transposing the matrices. This way, the index 'i' in the inner loop, is only added instead of multiplied to calculate the correct mapping of the element in the matrix to the array-represented matrix. 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 transposition. 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. The original versions, the sequential and the stock parallel, calculate these constant 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. By applying these techniques we could ran our parallel version around 10.7 times faster than sequential version and 3.9 times faster than the stock parallel version.

```
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];
```

Listing 2.3: Array-represented matrices transposition

```
# 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 2.4: Our parallel source code for sgemm

2.2 stencil

```
int i, j, k;
for(i=1;i<nx-1;i++) {
  for(j=1;j<ny-1;j++) {
    for(k=1;k<nz-1;k++) {
      Anext[Index3D (nx, ny, i, j, k)] =
      (A0[Index3D (nx, ny, i, j, k + 1)] +
      A0[Index3D (nx, ny, i, j, k - 1)] +
      A0[Index3D (nx, ny, i, j + 1, k)] +
      A0[Index3D (nx, ny, i, j - 1, k)] +
      A0[Index3D (nx, ny, i + 1, j, k)] +
      A0[Index3D (nx, ny, i - 1, j, k)])*c1
      - A0[Index3D (nx, ny, i, j, k)]*c0;
    }
  }
}
```

Listing 2.5: Parboil sequential source code for stencil

```
int i;
#pragma omp parallel for
for(i=1;i<nx-1;i++) {
  int j, k;
  for(j=1;j<ny-1;j++) {
    for(k=1;k<nz-1;k++) {
      //#pragma omp critical
      Anext[Index3D (nx, ny, i, j, k)] =
      (A0[Index3D (nx, ny, i, j, k + 1)] +
      A0[Index3D (nx, ny, i, j, k - 1)] +
      A0[Index3D (nx, ny, i, j + 1, k)] +
      A0[Index3D (nx, ny, i, j - 1, k)] +
      A0[Index3D (nx, ny, i + 1, j, k)] +
      A0[Index3D (nx, ny, i - 1, j, k)])*c1
      - A0[Index3D (nx, ny, i, j, k)]*c0;
    }
  }
}
```

Listing 2.6: Parboil parallel source code for stencil

```
#define Index3D(_nx,_ny,_i,_j,_k) ((_i)+_nx*((_j)+_ny*(_k)))
```

Listing 2.7: Converts from 3D indices to array-represented cube

The source codes above, specially the listing 2.7, show that the access to the cube elements are made by using the inner loop counter to multiply some constant and then map the cube to an array. The sequential access to memory is lost and the performance is impaired by sucessive cache misses.

Basically, we exchanged the index from the inner loop with the index of the outer loop. Then we simply parallelized the outer loop, which made our version run around 25.4 times faster than sequential version and around 15.2 times faster than the stock parallel version.

```

int i, j, k;
#pragma omp parallel for private(j, i)
for(k=1;k<nz-1;k++) {
  for(j=1;j<ny-1;j++) {
    for(i=1;i<nx-1;i++) {
      Anext[Index3D (nx, ny, i, j, k)] =
      (A0[Index3D (nx, ny, i, j, k + 1)] +
      A0[Index3D (nx, ny, i, j, k - 1)] +
      A0[Index3D (nx, ny, i, j + 1, k)] +
      A0[Index3D (nx, ny, i, j - 1, k)] +
      A0[Index3D (nx, ny, i + 1, j, k)] +
      A0[Index3D (nx, ny, i - 1, j, k)])*c1
      - A0[Index3D (nx, ny, i, j, k)]*c0;
    }
  }
}
```

Listing 2.8: Our parallel source code for stencil

2.3 mri-gridding

```
  unsigned int k;
  for(k=0; k<size; ++k){
    // compute value to evaluate kernel at
    // v in the range 0:(_width/2)^2
    v = (((float)k)/((float)size))*cutoff2;

    // compute kernel value and store
    (*LUT)[k] = kernel_value_CPU(beta*sqrt
        (1.0-(v/cutoff2)));
  }
```

Listing 2.9: Auxiliary function source code for sequential mri-gridding

```
  unsigned int k;
  #pragma omp parallel for private(v)
  for(k=0; k<size; ++k){
    // compute value to evaluate kernel at
    // v in the range 0:(_width/2)^2
    v = (((float)k)/((float)size))*cutoff2;

    // compute kernel value and store
    (*LUT)[k] = kernel_value_CPU(beta*sqrt
        (1.0-(v/cutoff2)));
  }
```

Listing 2.10: Auxiliary function source code for parallel mri-gridding

```
int i;
for (i=0; i < n; i++){
  ReconstructionSample pt = sample[i];

  float kx = pt.kX;
  float ky = pt.kY;
  float kz = pt.kZ;
```

Listing 2.11: Parboil sequential main loop for mri-gridding

```
int i;

#pragma omp parallel for private(NxL, NxH,
    NyL, NyH, NzL, NzH, dz2, nz, dx2, nx,
    dy2, ny, idxZ, idxY, dy2dz2, idx0, v,
    idx, w)
for (i=0; i < n; i++){
  ReconstructionSample pt = sample[i];

  float kx = pt.kX;
  float ky = pt.kY;
  float kz = pt.kZ;
```

Listing 2.12: Parboil parallel main loop for mri-gridding

```
    /* grid data */
    gridData[idx].real += (w*pt.real);
    gridData[idx].imag += (w*pt.imag);

    /* estimate sample density */
    sampleDensity[idx] += 1.0;
  }
```

Listing 2.13: Parboil sequential data processing code for mri-gridding

```
    /* grid data */
    #pragma omp critical (c1)
    gridData[idx].real += (w*pt.real);
    #pragma omp critical (c2)
    gridData[idx].imag += (w*pt.imag);

    /* estimate sample density */
    #pragma omp critical (c3)
    sampleDensity[idx] += 1.0;
```

Listing 2.14: Parboil parallel data processing code for mri-gridding

For this algorithm we basically allocated the lock-free structure and the accumulations are made privately and without concurrency for each thread. When all threads have finished the work, the final result is obtained by accumulating the partial results from each thread. Luckly, there is no loop-carried dependency, and the partial and the final accumulations are high parallelizable.

Our parallel version, with the lock-free vector accumulation, have a execution time around 2.5 times faster than the sequential and around 41.6 times faster than the stock parallel version. This happens because of the critical section pragmas, as we can see on listing 2.14. Each thread must execute this regions with a lock to obtain the correct result. This way, all other threads wait to get the lock and enter the critical region. By using this lock with 8 threads, there is a lot of concurrency and racing conditions, making the stock parallel the slowest version we tested, slower even the sequential version.

```

#pragma omp parallel private(NxL, NxH, NyL, NyH, NzL, NzH, dz2, nz, dx2,          \
  nx, dy2, ny, idxZ, idxY, dy2dz2, idx0, v, idx, w, t_id)
{
  #pragma omp single
  {
    if (big_table == NULL)
    {
      //          printf("Alocou!\n");
      size = params.gridSize[0]*params.gridSize[1]*params.gridSize[2];
      numthreads = omp_get_max_threads();
      big_table = (cmplx **) malloc(numthreads * sizeof(cmplx *));
      big_table2 = (float **) malloc(numthreads * sizeof(float *));
    }
  }

  #pragma omp for
  for (i = 0; i < numthreads; i++)
  {
    big_table[i] = (cmplx *) malloc(size * sizeof(cmplx));
    big_table2[i] = (float *) malloc(size * sizeof(float));
  }
  #pragma omp for collapse(2)
  for (i = 0; i < numthreads; i++)
    for (j = 0; j < size; j++)
    {
      big_table[i][j].real = 0;
      big_table[i][j].imag = 0;
      big_table2[i][j] = 0;
    }

  t_id = omp_get_thread_num();

  #pragma omp for
  for (i = 0; i < n; i++)
  {
    ReconstructionSample pt = sample[i];

    float kx = pt.kX;
    float ky = pt.kY;
    float kz = pt.kZ;
```

Listing 2.15: Our parallel lock-free structure allocation for mri-gridding

```
  /* grid data */
  big_table[t_id][idx].real += (w*pt.real);
  big_table[t_id][idx].imag += (w*pt.imag);

  /* estimate sample density */
  big_table2[t_id][idx] += 1.0;
```

Listing 2.16: Our parallel partial data processing code for mri-gridding

```
#pragma omp for private(j)
for (i = 0; i < size; i++)
  for (j = 0; j < numthreads; j++)
  {
    gridData[i].real += big_table[j][i].real;
    gridData[i].imag += big_table[j][i].imag;
    sampleDensity[i] += big_table2[j][i];
  }
```

Listing 2.17: Our parallel final data processing code for mri-gridding