Strassen算法多线程实现ver2

区别

与之前的实现的区别是，增大了并行的粒度，只有递归计算时，创建新线程处理

子任务

```
static class Task implements Callable<Matrix> {
    private Matrix A, B;
    private Section sa, sb;
    private int n;

    public Task(Matrix A, Matrix B, Section sa, Section sb, int n) {
        this.A = A;
        this.B = B;
        this.sa = sa;
        this.sb = sb;
        this.n = n;
    }

    @Override
    public Matrix call() throws Exception {
        return RecursiveSolve(A,B,sa,sb,n);
    }
}
```

核心函数

```
private static Matrix RecursiveSolve(Matrix A, Matrix B, Section sectionA,
Section sectionB, int n) throws ExecutionException, InterruptedException {
    //n小于等于64时，直接计算比递归更快
    if (n <= 128) {
        return Matrix.multiply(A, B, sectionA, sectionB, n);
    } else {
        Matrix[] S = new Matrix[10];
        Matrix[] P = new Matrix[7];
        //分解为n/2的子矩阵运算，后面运算的阶数都为mid
        int mid = n >> 1;
        //利用一个横纵坐标信息和阶数标记子矩阵
        Section A11 = new Section(sectionA.row, sectionA.column);
        Section A12 = new Section(sectionA.row, sectionA.column + mid);
        Section A21 = new Section(sectionA.row + mid, sectionA.column);
        Section A22 = new Section(sectionA.row + mid, sectionA.column +
mid);
        Section B11 = new Section(sectionB.row, sectionB.column);
        Section B12 = new Section(sectionB.row, sectionB.column + mid);
        Section B21 = new Section(sectionB.row + mid, sectionB.column);
        Section B22 = new Section(sectionB.row + mid, sectionB.column +
mid);
        Section S11 = new Section(0, 0);
        S[0] = Matrix.minus(B, B, B12, B22, mid);
```

```
        S[1] = Matrix.add(A, A, A11, A12, mid);
        S[2] = Matrix.add(A, A, A21, A22, mid);
        S[3] = Matrix.minus(B, B, B21, B11, mid);
        S[4] = Matrix.add(A, A, A11, A22, mid);
        S[5] = Matrix.add(B, B, B11, B22, mid);
        S[6] = Matrix.minus(A, A, A12, A22, mid);
        S[7] = Matrix.add(B, B, B21, B22, mid);
        S[8] = Matrix.minus(A, A, A11, A21, mid);
        S[9] = Matrix.add(B, B, B11, B12, mid);
        Future<Matrix> futureP0 = es.submit(new Task(A, S[0], A11, S11,
mid));
        Future<Matrix> futureP1 = es.submit(new Task(S[1], B, S11, B22,
mid));
        Future<Matrix> futureP2 = es.submit(new Task(S[2], B, S11, B11,
mid));
        Future<Matrix> futureP3 = es.submit(new Task(A, S[3], A22, S11,
mid));
        Future<Matrix> futureP4 = es.submit(new Task(S[4], S[5], S11, S11,
mid));
        Future<Matrix> futureP5 = es.submit(new Task(S[6], S[7], S11, S11,
mid));
        Future<Matrix> futureP6 = es.submit(new Task(S[8], S[9], S11, S11,
mid));
        //拿到P的结果
        P[0] = futureP0.get();
        P[1] = futureP1.get();
        P[2] = futureP2.get();
        P[3] = futureP3.get();
        P[4] = futureP4.get();
        P[5] = futureP5.get();
        P[6] = futureP6.get();
        Matrix[][] C = new Matrix[2][2];
        C[0][0] = new Matrix(mid);
        C[0][0].add(P[4]);
        C[0][0].add(P[3]);
        C[0][0].minus(P[1]);
        C[0][0].add(P[5]);
        C[0][1] = new Matrix(mid);
        C[0][1].add(P[0]);
        C[0][1].add(P[1]);
        C[1][0] = new Matrix(mid);
        C[1][0].add(P[2]);
        C[1][0].add(P[3]);
        C[1][1] = new Matrix(mid);
        C[1][1].add(P[4]);
        C[1][1].add(P[0]);
        C[1][1].minus(P[2]);
        C[1][1].minus(P[6]);
        return merge(C[0][0], C[0][1], C[1][0], C[1][1], mid);
    }
}
```