Laboratoire # 10 — Manipulation de polynomes en TBB/C++

a. Extraits du fichier `polynomes.c`

```
Polynome plus( Polynome p1, Polynome p2 )
{
  ordonnerPetitGrand( p1, p2 );
  assert( p1.degre <= p2.degre );

  Polynome p = allouer( p2.degre );

  if ( sequentiel() ) {
    for( size_t i = 0; i <= p1.degre; i++ ) {
      p.coefficients[i] = p1.coefficients[i] + p2.coefficients[i];
    }

    for( size_t i = p1.degre+1; i <= p2.degre; i++ ) {
      p.coefficients[i] = p2.coefficients[i];
    }
  } else {
    parallel_for( blocked_range<size_t>(0, p1.degre+1),
                  [=]( blocked_range<size_t> r ) {
                    for( size_t i = r.begin(); i < r.end(); i++ ) {
                      p.coefficients[i] = p1.coefficients[i] + p2.coefficients[i];
                    }
                  }
                  );
    parallel_for( blocked_range<size_t>(p1.degre+1, p2.degre+1),
                  [=]( blocked_range<size_t> r ) {
                    for( size_t i = r.begin(); i < r.end(); i++ ) {
                      p.coefficients[i] = p2.coefficients[i];
                    }
                  }
                  );
  }

  return p;
}
```

```
static double coefficient( size_t k, Polynome p1, Polynome p2 )
{
  size_t expMinR1 = k < p2.degre ? 0 : k-p2.degre;
  size_t expMaxR1 = k < p1.degre ? k : p1.degre;

  assert( expMinR1 <= expMaxR1 && "expMinR1 doit etre <= expMaxR1??" );

  if ( sequentiel() ) {
    double c = 0.0;
    for( size_t i = expMinR1; i <= expMaxR1; i++ ) {
      c += p1.coefficients[i] * p2.coefficients[k-i];
    }
    return c;
  } else {
    return
      parallel_reduce( blocked_range<size_t>(expMinR1, expMaxR1+1),
                       0.d,
                       [=]( blocked_range<size_t> r, double c ) {
                         for( size_t i = r.begin(); i < r.end(); i++ ) {
                           c += p1.coefficients[i] * p2.coefficients[k-i];
                         }
                         return c;
                       },
                       std::plus<double>()
                       );
  }
}

//
```

```
Polynome fois( Polynome p1, Polynome p2 )
{
  Polynome p = allouer( p1.degre + p2.degre );

  if ( sequentiel() ) {
    for( size_t i = 0; i <= p.degre; i++ ) {
      p.coefficients[i] = coefficient( i, p1, p2 );
    }
  } else {
    parallel_for( blocked_range<size_t>(0, p.degre+1),
                  [=]( blocked_range<size_t> r ) {
                    for( size_t i = r.begin(); i < r.end(); i++ ) {
                      p.coefficients[i] = coefficient( i, p1, p2 );
                    }
                  }
                  );
  }

  return p;
}
```

```
double eval( Polynome p, double x )
{
  if ( sequentiel() ) {
    double r = 0.0;
    for( size_t i = 0; i <= p.degre; i++ ) {
      r += p.coefficients[i] * std::pow(x, i);
    }
    return r;
  } else {
    return
      parallel_reduce( blocked_range<size_t>(0, p.degre+1),
                       0.d,
                       [=]( blocked_range<size_t> r, double val ) {
                         for( size_t i = r.begin(); i < r.end(); i++ ) {
                           val += p.coefficients[i] * std::pow(x, i);
                         }
                         return val;
                       },
                       std::plus<double>()
                       );
  }
}
```

```
bool egaux( Polynome p1, Polynome p2 )
{
  ordonnerPetitGrand( p1, p2 );
  assert( p1.degre <= p2.degre );

  if ( sequentiel() ) {
    for( size_t i = 0; i <= p1.degre; i++ ) {
      if ( p1.coefficients[i] != p2.coefficients[i] )
        return false;
    }

    for( size_t i = p1.degre+1; i <= p2.degre; i++ ) {
      if ( p2.coefficients[i] != 0 )
        return false;
    }

    return true;
  } else {
    bool res = true;
    task_group gr;

    for( size_t i = 0; i <= p1.degre; i++ ) {
      gr.run( [i,p1,p2,&gr,&res] {
          if ( p1.coefficients[i] != p2.coefficients[i] ) { res = false; gr.cancel()
        } );
    };

    for( size_t i = p1.degre+1; i <= p2.degre; i++ ) {
      gr.run( [i,p2,&gr,&res] {
          if ( p2.coefficients[i] != 0 ) { res = false; gr.cancel(); }
        } );
    }

    gr.wait();
    return res;
  }
}
```

b. Temps d'exécution pour diverses tailles de polynomes

Temps d'exécution (sur `japet`) pour des polynomes avec 7 000 vs. 8 000 coefficients, où on obtient des accélérations lorsqu'on utilise 16 ou 32 *threads* avec 8 000 coefficients — mais l'accélération augmente lorsqu'on a encore plus de coefficients à traiter :

```
$ make mesures N=7000
./mesurer-polynomes 7000
# N = 7000
# nb.th.  seq        par
  1       0.293      0.782
  2       0.293      0.525
  4       0.293      0.413
  8       0.293      0.350
 16       0.293      0.330
 32       0.293      0.342
 64       0.293      0.352
$ make mesures N=8000
./mesurer-polynomes 8000
# N = 8000
# nb.th.  seq        par
  1       0.403      0.937
  2       0.403      0.625
  4       0.403      0.487
  8       0.403      0.417
 16       0.403      0.394
 32       0.403      0.397
 64       0.403      0.494


$ make mesures N=20000
./mesurer-polynomes 20000
# N = 20000
# nb.th.  seq        par
  1       2.385      4.543
  2       2.385      2.685
  4       2.385      1.753
  8       2.385      1.331
 16       2.385      1.139
 32       2.385      1.110
 64       2.385      1.187
```

c. Temps d'exécution avec l'ancienne version
Le temps d'exécution est **beaucoup** plus long.

```
./mesurer-polynomes 20000
# N = 20000
# nb.th.  seq        par
  1       2.441    187.791
  2       2.441     92.140
  4       2.441     46.783
  8       2.441     25.156
 16       2.441     13.027
 32       2.441      7.264
 64       2.441      6.041
```