Matrix-vector multiplication with QtConcurrent :: mappedReduced()

This article presents matrix-vector multiplication using the MapReduce programming model with QtConcurrent.

Variant 1

qtconcurrentmatrixvector/main.cpp

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector/main.cpp)

qtconcurrentmatrixvector/sparsematrix.txt

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector/sparsematrix.txt)

qtconcurrentmatrixvector/vector.txt

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector/vector.txt)

qtconcurrentmatrixvector/qtconcurrentmatrixvector.pro

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector/qtconcurrentmatrixvector.pro) (Qt 4 compatible)

qtconcurrentmatrixvector/qtconcurrentmatrixvector.pro

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector/pro_file_qt5/qtconcurrentmatrixvector.pro) (Qt 5 compatible)

Variant 2

qtconcurrentmatrixvector2/main.cpp

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector2/main.cpp)

qtconcurrentmatrixvector2/sparsematrix.txt

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector2/sparsematrix.txt)

qtconcurrentmatrixvector2/vector.txt

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector2/vector.txt)

qtconcurrentmatrixvector2/qtconcurrentmatrixvector2.pro

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector2/qtconcurrentmatrixvector2.pro) (Qt 4 compatible)

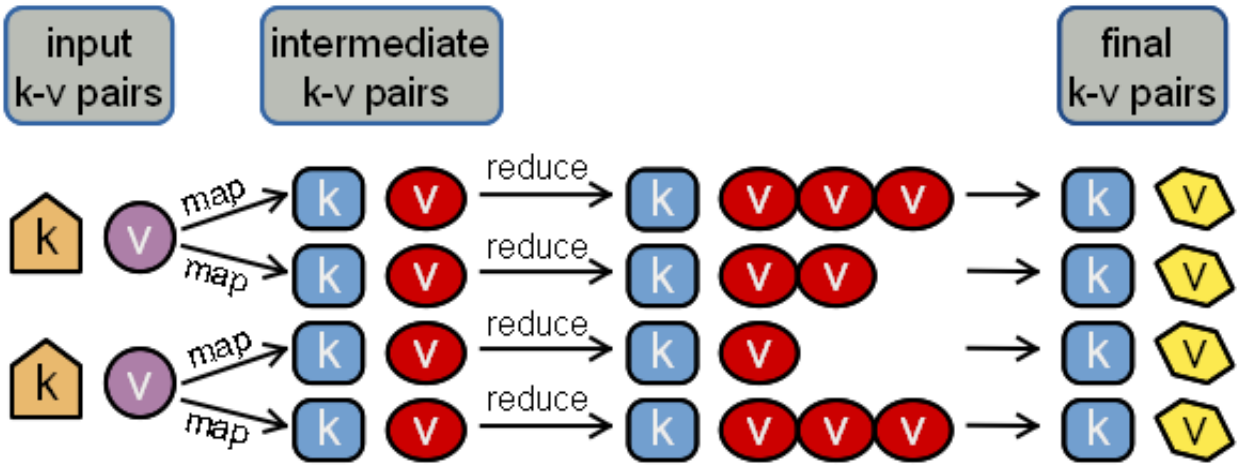
qtconcurrentmatrixvector2/qtconcurrentmatrixvector2.pro

(qtconcurrentmatrixvectorSource/qtconcurrentmatrixvector2/pro_file_qt5/qtconcurrentmatrixvector2.pro) (Qt 5 compatible)

Introduction - MapReduce

MapReduce is a high level programming model for processing large data sets in parallel, originally developed by Google, adapted from functional programming. The model is suitable for a range of problems such as matrix operations, relational algebra, statistical frequency counting etc. To learn more about MapReduce, you can read Chapter 2 from the book Mining of Massive Datasets by Lescovec, Rajaraman & Ullman. (http://infolab.stanford.edu/~ullman/mmds/ch2.pdf)

In general, **a map function takes an input and produces an intermediate key-value pair** (as in this example) or a list of intermediate key-value pairs. **Reduce function performs a summary operation on the intermediate keys.** You can see this below:



The diagram has been inspired by a course presentation Massive Parallelism with MapReduce (http://delab.csd.auth.gr/courses/c_dm_pms/MR_pms_dm2.pdf) and adjusted to the needs of the present implementation.

Firstly, a short reminder of how matrix-vector multiplication works (figure below). To calculate a specific entry of the matrix-vector product we have to multiply corresponding row entries of the matrix by the components of the vector and sum up the values. To get the first component of the matrix-vector product, we have to multiply the elements in the first row of the matrix by the corresponding vector components and sum up the values. In matrix-vector multiplication, the number of columns of the matrix has to be equal to the number of components of the vector.

The MapReduce implementation in this example differs from the school-book multiplication that I just introduced. A single map function will be processing only a single matrix element rather than the whole row.

$$\begin{bmatrix} 3 & 2 & 0 \\ 0 & 4 & 1 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 18 \\ 13 \\ 9 \end{bmatrix}$$

> *There are different implementations of matrix vector multiplication depending on whether the vector fits into the main memory or not.* ***This example demonstrates a basic version of matrix-vector multiplication in which the vector fits into the main memory.***

We store **the sparse matrix** (sparse matrices often appear in scientific and engineering problems) **as a triple with explicit coordinates (i, j, a_{ij})**. E.g. the value of the first entry of the matrix (0,0) is 3. Similarly, the value of the second entry in (0,1) is 2. We do not store the zero entries of the matrix. This is the sparse matrix representation of the matrix from the figure above:

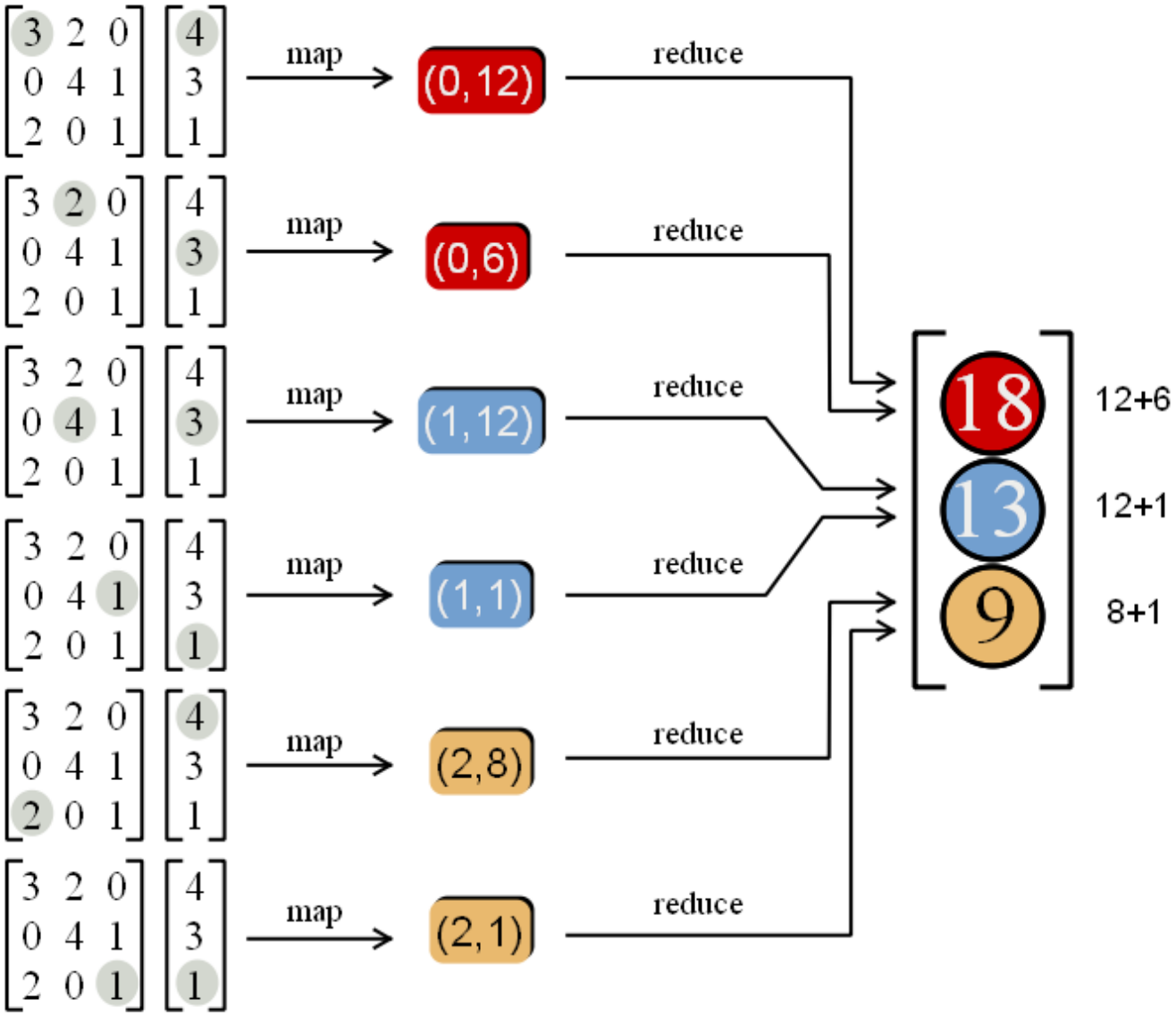
```
i, j, aij
0,0,3
0,1,2
1,1,4
1,2,1
2,0,2
2,2,1
```

We store the vector in a dense format without explicit coordinates. You can see this below:

```
4
3
1
```

In our case, the map function takes a single element of the sparse matrix (not the whole row!), multiplies it with the corresponding entry of the vector and produces an intermediate key-value pair (i, a_{ij}*v_j). This is sufficient because in order to perform the summation (i.e. the reduce step) we only need to know the matrix row, we do not need to know the matrix column. E.g. one of the map functions takes the first element of the matrix (0,0,3), multiplies it with the first element of the vector (4) and produces an intermediate key-value pair (0,12). The key (0), i.e. the row position of the element in the sparse matrix, associates the value (12) with its position in the matrix-vector product. Another map function takes the second element (0,1,2), multiplies it with the second row of the vector (3) producing an intermediate key-value pair (0,6) etc.

Reduce function performs a summary operation on the intermediate keys. E.g. an intermediate value with the key "0" will be summed up under a final index "0".



General implementation for both variants - without and with functor

Two variants of the implementation will be presented in the following paragraphs:

- **without a functor** - the input vector is a global variable
- **with a functor** - the input vector is stored locally and passed as a parameter

The following implementation is identical for both versions: We read the elements of the sparse matrix and the vector from separate files, namely 'sparsematrix.txt' and 'vector.txt'. In order to read them, we use the following functions:

```
bool populateMatrix(QString fileName, QStringList& sparseMatrix)
bool populateVector(QString fileName, QVector<double>& vector)
```

The first function takes an empty string list and populates it with the values from the file 'sparsematrix.txt'. One string in the sparseMatrix list represents one matrix element. Similarly, the second function populates an empty vector with the values from the file 'vector.txt'. Both functions check the existence of the files and validity of the values.

```
bool populateMatrix(QString fileName, QStringList& sparseMatrix) {
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)){
        qDebug("Matrix file could not be opened.");
        return false;
    }
    QTextStream in(&file);
    while (!in.atEnd()) {
        QString line = in.readLine();
        QStringList splitLine = line.split(",");
        if(splitLine.size() != 3){
            qDebug() << "Invalid matrix format.";
            return false;
        }
        bool iOk = false;
        bool jOk = false;
        bool aijOk = false;
        splitLine.at(0).toUInt(&iOk);
        splitLine.at(1).toUInt(&jOk);
        splitLine.at(2).toDouble(&aijOk);
        if(iOk && jOk && aijOk){
            sparseMatrix.append(line);
        }
        else {
            qDebug() << "Invalid matrix element.";
            return false;
        }
    }
    return true;
}
```

```
bool populateVector(QString fileName, QVector<double>& vector){
    QFile file(fileName);

    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)){
        qDebug() << "Vector file could not be opened.";
        return false;
    }
    QTextStream in(&file);
    while (!in.atEnd()) {
        QString line = in.readLine();
        bool valueOk = false;
        double row = line.toDouble(&valueOk);
        if(valueOk)
            vector.append(row);
        else {
            qDebug() << "Invalid vector element.";
            return false;
        }
    }
    return true;
}
```

Variant 1 - without function object

The first variant presents a simpler solution which takes advantage of a map function. The second variant uses a map function object (functor) instead of a map function.

Qt's mappedReduced requires three parameters - a sequence to which to apply the map function, a map function and a reduce function. The fourth parameter is optional and specifies the order in which results from the map function are passed to the reduce function (not relevant for our implementation). You can see the signature below:

> ***QFuture<T> QtConcurrent::mappedReduced****(const Sequence & sequence, MapFunction mapFunction, ReduceFunction reduceFunction, QtConcurrent::ReduceOptions reduceOptions = UnorderedReduce | SequentialReduce)*

Let's look at the main() below. We firstly populate the sparse matrix and the vector with the values from the files. Notice, that unlike the sparseMatrix, **the input vector is a global variable - this is because every map function needs to have access to the vector** (this is precisely how variant 2 differs from variant 1, in variant 2 we are going to use map function object to pass the vector as a parameter rather than providing global vector access).

```
QVector<double> vec;

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QStringList sparseMatrix;
    bool matrixOk = populateMatrix("sparsematrix.txt", sparseMatrix);
    bool vectorOk = populateVector("vector.txt", vec);

    if(matrixOk && vectorOk){
        QFuture< QVector<double> > f1;
        f1 = QtConcurrent::mappedReduced(sparseMatrix, mapElement, reduce);
        f1.waitForFinished();
        qDebug() << f1.result();
    }
    return 0;
}
```

After populating the matrix and the vector we create **an instance of QFuture that will represent the result of our calculation**. After that we call mappedReduced. We wait until the asynchronous computation finishes and print out the result.

The map function is called for every sparse matrix entry, i.e. for every string in the QStringList, in our case the map function will be called 6 times. A map function takes a single matrix element, multiplies it with the corresponding vector element and returns a pair (matrix element row index i, result of the multiplication $a_{ij}*v_j$). **Map functions are called concurrently, which means they have to be written as thread safe (in this case irrelevant).**

```
QPair<uint, double> mapElement(const QString& element) {
    QStringList splitElement = element.split(",");
    uint i = splitElement.at(0).toUInt();
    uint j = splitElement.at(1).toUInt();
    double aij = splitElement.at(2).toDouble();
    return qMakePair(i, aij*vec.at(j));
}
```

> ***uint** is Qt's typedef for **unsigned int, i.e quint32**, holding values from **0 to 4,294,967,295** which equals to $2^{32} - 1$.*

The reduce function will perform a summation on the values produced by the map functions for a given key i. The first parameter of the reduce function is the vector in which we are going to store our result (this is a non-const reference because we are going to update this vector). The second parameter is the intermediate result produced by the map function. What the reduce function does, is update the matrix-vector product on the i^{th} position (position given by pair.first) by the value given by pair.second. **Notice, that unlike with the map function, only one thread at a time will call the reduce function, which means that a synchronisation mechanism (e.g. mutex) does not have to be used.**

```
void reduce(QVector<double> &product,
                              const QPair<uint,double> &pair) {
    if(product.isEmpty())
        product.resize(vec.size());
    product[pair.first] += pair.second;
}
```

Variant 2 - with a function object

In the second variant, we are going to use a map function object instead of a map function. This is so that we can pass the input vector as a reference, rather than storing it as a global variable. To accomplish this, we only need to make a couple of changes to the main() and change the mapElement() function to a functor.

```
uint vec_size;

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    QStringList sparseMatrix;
    bool matrixOk = populateMatrix("sparsematrix.txt", sparseMatrix);
    QVector<double> vec;
    bool vectorOk = populateVector("vector.txt", vec);

    if(matrixOk && vectorOk){
        vec_size = vec.size();
        QFuture< QVector<double> > f1;
        f1 = QtConcurrent::mappedReduced(sparseMatrix, MapElement(&vec),
                                         reduce);
        f1.waitForFinished();
        qDebug() << f1.result();
    }
    return 0;
}
```

Notice that, we now no longer store the vector as a global variable. The input vector is introduced in the main(). We still have to store the size of the input vector as a global variable in order to resize the matrix-vector product in the reduce function. Also notice that we are no longer passing the map function but rather the MapElement function object with a reference to the input vector as a parameter.

Now let's look at the map function object. **A function object or a functor is simply an object that can be called as if it was a function. To achieve this, it defines operator()**. Look at the code snippet below. You see that the body of the overloaded operator() is almost identical to the mapElementl() function from the Variant 1. We also store the pointer to the vector as a member variable. This variable will be assigned through the constructor (look at the mappedReduced in the main()). Last, for our functor to work, we also have to define the **result_type** of the function call operator (which is QPair<uint, double> in this case).

```
struct MapElement {
    const QVector<double> *m_vec;
    MapElement(const QVector<double> *vec): m_vec(vec) {}
    typedef QPair<uint, double> result_type;

    QPair<uint, double> operator()(const QString& element)
    {
        QStringList splitElement = element.split(",");
        uint i = splitElement.at(0).toUInt();
        uint j = splitElement.at(1).toUInt();
        double aij = splitElement.at(2).toDouble();
        return qMakePair(i, aij*m_vec->at(j));
    }
};
```

Last, let's look at the reduce function. The only change with respect to the Variant 1 is that I resize the matrix-vector product by the global variable vec_size.

```
void reduce(QVector<double> &product,
                            const QPair<uint,double> &pair) {
    if(product.isEmpty())
        product.resize(vec_size);
    product[pair.first] += pair.second;
}
```

> *That's it! You can now try out your concurrent calculation. You should get the result* ***QVector(18, 13, 9)****.*

Latest update: (../course/qtconcurrentmatrixvectorSource/qtconcurrentmatrixvectorUpdates.txt)06.07.2016
Created: 2015