MOPL
Matrix Operation Parallelism Language

Designed and implemented by:
Fengwei Zhang,
Maoliang Huang,
Thomas Ricatte,
Jorge Portal,
Seoho Lee

April 29th
Motivation

- Explosion of multiprocessor technology in modern times
- Software designers should (and will be required to soon, if not already) take advantage of this technology
  - Stay competitive
  - Better quality of software
  - Higher user satisfaction
- Researchers can definitely benefit from exploiting this technology as well
  - Less time-consuming modeling and experimentation compared to uniprocessor
  - Access to a relative abundance of computational resources
- Generally clunky, confusing, and difficult to engineer software that
What about a programming language that resolves around exploiting this technology, while making the job of dividing software into multiple processors almost invisible to the programmers?

Enter MOPL
MOPL at a glance

- MOPL is the Matrix Operation Parallelism Language.
- MOPL is designed to perform lightning-fast, safe, and multithreaded operations, focusing its power primarily in the domain of matrices.
  - Matrix addition, matrix multiplication, etc.
- Multiprocessor programming is invisible to the programmer in MOPL. MOPL uses an internal system to decide how to divide segments of operations into threads and how many.
- MOPL utilizes a "divide-and-conquer" approach to matrix operations, delegating responsibility for submatrices of the operations to the threads it creates, so they execute in parallel.
- Result: highly scalable and rapid matrix operations to suit the end-users’ needs, whether it is a corporation, researcher, or consumer.
MOPL Architecture

Architecture

Source code

Scanner

Symbol table

Parser

Error handler

Syntax tree

Code Generator

`public static void main(String[] args) {
    Matrix E = new Matrix(2, 2, "{2, 3; 4, 2");
    Matrix D = new Matrix(2, 2, "{1, 2; 3, 2");
    C = new Matrix(2, 2);
    C = MopiOperations.multiply(D, E);
    System.out.print("result:" + C.toString());
}"

External libraries

JVM

Java compiler

Java bytecode

Output
main()
{
    scalar i;
    // declare a 3x3 square matrix
    const matrix b(3,3) = {2,3,7:9,2,3:3,4,2};

    // declare an array of matrix.
    // The size of the array is 3
    // and each matrix is a 3x3 square matrix.
    matrix[3] c(3,3);

    for (i=0; i<3; i++) {
        c[i] = {1,0,0:0,1,0:0,0,1};
    }

    c[0] = c[0] * b;
    i = 10;
    c[1] = i * b;
    c[2] = c[0] - c[1];
}
Tools

- SVN repository, Wiki, ...
- Awesome IDE for PLT projects
- JVM, javacc, jjtree
- Used for the internal mailing-list
Parallelism features

- Parallelism is user-invisible
  - External Matrix Operation Library
  - Parallel matrix addition, subtraction, multiplication functions in the library

- Implementation
  - Using multi-threads running simultaneously do matrix operations
Threads Testing Experiment

- **50x50 multiplication - data for 1-50 threads**
- **200x200 multiplication - data for 1-100 threads**
- **500x500 multiplication - data for 100-300 threads**
- **Observed creation time per thread**
Beyond certain threads, there is no gain at all because the processor is saturated with work.

Adding another thread will only give it another task to do that it still has to be scheduled.

On our Core 2 laptop, there is a big difference between liner running and two threads running.

Slight changes if we add more threads
Matrix multiplication algorithm

```
MULT(C,A,B, n)
1 if n= bs (block size)
3 return
4 allocate a temporary matrix T[1..n,1..n]
5 partition A, B, C, and T into (n/2)x(n/2) submatrices
6 spawn MULT(C11,A11,B11,n/2)
7 spawn MULT(C12,A11,B12,n/2)
8 spawn MULT(C21,A21,B11,n/2)
9 spawn MULT(C22,A21,B12,n/2)
10 spawn MULT(T11,A12,B21,n/2)
11 spawn MULT(T12,A12,B22,n/2)
12 spawn MULT(T21,A22,B21,n/2)
13 spawn MULT(T22,A22,B22,n/2)
14 sync
15 ADD(C,T,n)
```

- Divide and Conquer algorithm
- Divide the matrix into 4 sub matrices
- Do recursion
- Until : sub-matrices size < block size
Test plan

- Correctness, completeness, and efficiency.
- MOPL should run every input correctly.
- MOPL should check on every grammar
- MOPL should effectively implement parallelism, and take the advantage of multi-threading for faster computation
Test plan

- **Test 1**: Phase to assess our algorithm before final version.
  - White box testing each on our Java Class
  - Effectiveness of our threading
  - Regression testing to find software regressions.
- **Test 2**: Testing after our final program to assess our language
  - Black box testing
  - Test Suites to cover simple test model which test our efficiency and advanced model to test our correctness
Conclusion

- Parallelism
- Multithreading is difficult to implement
- Finding Effective Matrix computation.
- JavaCC
- MOPL is a simple and fast program that computes Matrix faster than any other language