On the Performance of Parametric Polymorphism in Maple

Laurentiu Dragan    Stephen M. Watt
Ontario Research Centre for Computer Algebra
University of Western Ontario

Maple Conference 2006
Outline

- Parametric Polymorphism
- SciMark
- SciGMark
- A Maple Version of SciGMark
- Results
- Conclusions
Parametric Polymorphism

- **Type Polymorphism** – Allows a single definition of a function to be used with different types of data

- **Parametric Polymorphism**
  - A form of polymorphism where the code does not use any specific type information
  - Instances with type parameters

- Increasing popularity – C++, C#, Java

- Code reusability and reliability

- Generic Libraries – STL, Boost, NTL, LinBox, Sum-IT (Aldor)
SciMark

- National Institute of Standards and Technology
  - http://math.nist.gov/scimark2
- Consists of five kernels:
  
  1. Fast Fourier transform
     - One-dimensional transform of 1024 complex numbers
     - Each complex number 2 consecutive entries in the array
     - Exercises complex arithmetic, non-constant memory references and trigonometric functions
2. **Jacobi successive over-relaxation**
   - 100 × 100 grid
   - Represented by a two dimensional array
   - Exercises basic “grid averaging” – each A(i, j) is assigned the average weighting of its four nearest neighbors

3. **Monte Carlo**
   - Approximates the value of π by computing the integral part of the quarter unit cycle
   - Random points inside the unit square – compute the ratio of those within the cycle
   - Exercises random-number generators, function inlining
4. **Sparse matrix multiplication**
   - Uses an unstructured sparse matrix representation stored in a compressed-row format
   - Exercises indirection addressing and non-regular memory references

5. **Dense LU factorization**
   - LU factorization of a dense $100 \times 100$ matrix using partial pivoting
   - Exercises dense matrix operations
SciMark

- The kernels are repeated until the time spent in each kernel exceeds a certain threshold (2 seconds in our case)
- After the threshold is reached, the kernel is run once more and timed
- The time is divided by number of floating point operations
- The result is reported in MFlops (or Million Floating-point instructions per second)
SciMark

- There are two sets of data for the tests: large and small
- Small uses small data sets to reduce the effect of cache misses
- Large is the opposite of small 😊
- For our Maple tests we used only the small data set
SciGMark

- Generic version of SciMark (SYNASC 2005)
  - http://www.orcca.on.ca/benchmarks
- Measure difference in performance between generic and specialized code
- Kernels rewritten to operate over a generic numerical type supporting basic arithmetic operations (+, -, ×, /, zero, one)
- Current version implements a wrapper for numbers using double precision floating-point representation
Parametric Polymorphism in Maple

- Module-producing functions
  - Functions that take one or more modules as arguments and produce modules as their result
  - Resulting modules use operations from the parameter modules to provide abstract algorithms in a generic form
Example

MyGenericType := proc(R)
    module ()
        export f, g;
        #Here f and g can use u and v from R
        f := proc(a, b) foo(R:-u(a), R:-v(b)) end;
        g := proc(a, b) goo(R:-u(a), R:-v(b)) end;
    end module:
end proc:
Approaches

- **Object-oriented**
  - Data and operations together
  - Module for each value
  - Closer to the original SciGMark implementation

- **Abstract Data Type**
  - Each value is some data object
  - Operations are implemented separately in a generic module
  - Same module shared by all the values belonging to each type
Object-Oriented Approach

DoubleRing := proc(val::float)
    local Me;
    Me := module()
        export v, a, s, m, d, gt, zero, one,
            coerce, absolute, sine, sqroot;
        v := val; # Data value of object
        # Implementations for +, -, *, /, >, etc
        a := (b) -> DoubleRing(Me:-v + b:-v);
        s := (b) -> DoubleRing(Me:-v - b:-v);
        m := (b) -> DoubleRing(Me:-v * b:-v);
        d := (b) -> DoubleRing(Me:-v / b:-v);
        gt := (b) -> Me:-v > b:-v;
        zero := () -> DoubleRing(0.0);
        coerce := () -> Me:-v;
        . . .
    end module:
    return Me;
end proc:
Object-Oriented Approach

- Previous example simulates object-oriented approach by storing the value in the module.
- The exports a, s, m, d correspond to basic arithmetic operations.
- We chose names other than the standard +, -, ×, / for two reasons:
  - The code looks similar to the original SciGMark (Java does not have operator overloading).
  - It is not very easy to overload operators in Maple.
- Functions like sine and sqrt are used by the FFT algorithm to replace complex operations.
Abstract Data Type Approach

DoubleRing := module()
    export a, s, m, d, gt, zero, one,
        coerce, absolute, sine, sqroot;
    # Implementations for +, -, *, /, >, etc
    a := (a, b) -> a + b;
    s := (a, b) -> a - b;
    m := (a, b) -> a * b;
    d := (a, b) -> a / b;
    gt := (a, b) -> a > b;
    zero := () -> 0.0;
    one := () -> 1.0;
    coerce := (a::float) -> a;
    absolute := (a) -> abs(a);
    sine := (a) -> sin(a);
    sqroot := (a) -> sqrt(a);
end module:
Abstract Data Type Approach

- Module does not store data, provides only the operations
- As a convention one must coerce the float type to the representation used by this module
- In this case the representation is exactly float
- DoubleRing module created only once for each kernel
Kernels

- Each SciGMark kernel exports an implementation of its algorithm and a function to compute the estimated floating point operations.
- Each kernel is parametrized by a module $R$, that abstracts the numerical type.
Kernel Structure

gFFT := proc(R)
    module()
    export num_flops, transform, inverse;
    local transform_internal, bitreverse;
    num_flops := . . .;
    transform := . . .;
    inverse := . . .;
    transform_internal := . . .;
    bitreverse := . . .;
    end module:
end proc:
The high level structure is the same for object-oriented and for abstract data type.

Implementation inside the functions is different.

<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialized</td>
<td>$x^2 + y^2$</td>
</tr>
<tr>
<td>Object-oriented</td>
<td>$(x:\text{\texttt{-}}m(x)\text{\texttt{:\texttt{-}}a(y:\text{\texttt{-}}m(y))))\text{\texttt{:\texttt{-}}coerce()}$</td>
</tr>
<tr>
<td>Abstract Data Type</td>
<td>$R:\text{\texttt{-coerce}}(R:\text{\texttt{-a}(R:\text{\texttt{-m}}(x,x), R:\text{\texttt{-m}}(y,y))))$</td>
</tr>
</tbody>
</table>
GenMonteCarlo := proc(DR::`module`)  
    local m;  
    m := module ()  
        export num_flops, integrate;  
        local SEED; SEED := 113;  
        num_flops := (Num_samples) -> Num_samples * 4.0;  
        integrate := proc (numSamples)  
            local R, under_curve, count, x, y, nsml;  
            R := Random(SEED);  
            under_curve := 0; nsml := numSamples - 1;  
            for count from 0 to nsml do  
                x := DR:-coerce(R:-nextDouble());  
                y := DR:-coerce(R:-nextDouble());  
                if DR:-coerce(DR:-a(DR:-m(x,x), DR:-m(y, y))) <= 1.0 then  
                    under_curve := under_curve + 1;  
                end if;  
                end if;  
            end do;  
            return (under_curve / numSamples) * 4.0;  
        end proc;  
    end module:  
    return m;  
end proc:
Kernel Sample (Object-Oriented)

GenMonteCarlo := proc(r::`procedure``)`
    local m;
    m := module ()
        export num_flops, integrate;
        local SEED; SEED := 113;
        num_flops := (Num_samples) -> Num_samples * 4.0;
        integrate := proc (numSamples)
            local R, under_curve, count, x, y, nsml;
            R := Random(SEED);
            under_curve := 0; nsml := numSamples - 1;
            for count from 0 to nsml do
                x := r(R:-nextDouble());
                y := r(R:-nextDouble());
                if (x:-m(x):-a(y:-m(y))):-coerce() <= 1.0 then
                    under_curve := under_curve + 1;
                end if;
            end do;
            return (under_curve / numSamples) * 4.0;
        end proc;
    end module:
    return m;
end proc:
measureMonteCarlo := proc(min_time, R)
    local Q, cycles;
    Q := Stopwatch();
    cycles := 1;
    while true do
        Q:-strt();
        GenMonteCarlo(DoubleRing):-integrate(cycles);
        Q:-stp();
        if Q:-rd() >= min_time then break; end if;
        cycles := cycles * 2;
    end do;
    return GenMonteCarlo(DoubleRing):-num_flops(cycles) / Q:-rd()
        * 1.0e-6;
end proc;
## Results (MFlops)

<table>
<thead>
<tr>
<th>Test</th>
<th>Specialized</th>
<th>Abstract Data Type</th>
<th>Object Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast Fourier Transform</td>
<td>0.123</td>
<td>0.088</td>
<td>0.0103</td>
</tr>
<tr>
<td>Successive Over Relaxation</td>
<td>0.243</td>
<td>0.166</td>
<td>0.0167</td>
</tr>
<tr>
<td>Monte Carlo</td>
<td>0.092</td>
<td>0.069</td>
<td>0.0165</td>
</tr>
<tr>
<td>Sparse Matrix Multiplication</td>
<td>0.045</td>
<td>0.041</td>
<td>0.0129</td>
</tr>
<tr>
<td>LU Factorization</td>
<td>0.162</td>
<td>0.131</td>
<td>0.0111</td>
</tr>
<tr>
<td>Composite</td>
<td>0.133</td>
<td>0.099</td>
<td>0.0135</td>
</tr>
<tr>
<td>Ratio</td>
<td>100%</td>
<td>74%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Note: Larger means faster
Results

- Abstract Data Type is very close in performance to the specialized version – about 75% as fast
- Object-oriented model simulates closely the original SciGMark – produces many modules and this leads to a significant overhead about only 10% as fast
- Useful to separate the instance specific data from the shared methods module – values are formed as composite objects from the instance and the shared methods module
Conclusions

- Performance penalty should not discourage writing generic code
  - Provides code reusability that can simplify libraries
  - Writing generic programs in mathematical context helps programmers operate at a higher level of abstraction

- Generic code optimization is possible and we proposed an approach to optimize it by specializing the generic type according to the instances of the type parameters
Conclusions (Contd.)

- Parametric polymorphism does not introduce excessive performance penalty
  - Possible because of the interpreted nature of Maple, not many optimizations performed on the specialized code (even specialized code uses many function calls)

- Object-oriented use of modules not well supported in Maple; simulating sub-classing polymorphism in Maple is very expensive and should be avoided

- Better support for overloading would help programmers write more generic code in Maple.

- More info about SciGMark at: http://www.orcca.on.ca/benchmarks/
Acknowledgments

- ORCCA members
- MapleSoft