Comparison of Parallel Computing in *Mathematica* 10 and Maple 2015

Maplesoft marketing literature claims that *Mathematica*'s parallel programming requires "complicated message-passing between nodes to communicate context and state. This style of parallel programming takes longer to develop and debug, and generally requires an experienced programmer."

This document will show that the reality is the complete reverse of this statement. *Mathematica* requires less programming to achieve a more robust and faster result.

To show this, we take a Maplesoft marketing example for its parallel programming tools (from www.maplesoft.com/products/maple/features/gridcomputing.aspx, presumed to be an example of best practice for Maple) and implement the same task in *Mathematica*.

Maple

The document starts by creating a simple Monte Carlo integrator called parallelApproxint (code for this is at the end of this document). Maple then parallelizes this procedure with the following code:

```
parallelApproxint := proc( expr, lim::name=numeric..numeric,
                           { numSamples::integer := 1000 })
    uses Grid;
    local me, numNodes, r, n;

    me := MyNode();
    numNodes := NumNodes();
    n := trunc(numSamples/numNodes);
    r:=approxint( expr, lim, numsamples = n );
    printf("Node %d compute result %a using %d samples\n", me, r, n);

    if me = 0 then
        r := (r+add( Receive(), i=1..numNodes-1 )) / numNodes;
    else
        Send(0,r):
    end if;
end proc:

Grid:-Setup("hpc");
Grid:-Launch(parallelApproxint, x^2,x=1..3,numSamples=10^7,
    imports=['approxint'], numnodes=16);
```

Mathematica

The *Mathematica* equivalent is much simpler:

```
parallelApproxInt[args__, n_Integer] := Mean[
   ParallelTable[approxInt[args, Floor[n / $KernelCount]], {$KernelCount}]];
parallelApproxInt[x^2, {x, 1, 3}, 10^7]
```

Key observations

There are sevaral important differences to note:

- Despite the Maplesoft claim that *Mathematica* requires complicated messaging, it is only the Maple version that requires the user to write explicit messaging commands (`Send` and `Receive`). Messaging between nodes is entirely automated within *Mathematica.*
- In the Maple code, the user must explicitly list functions that must be passed to the compute node (using `imports=['approxint']`). *Mathematica* automatically distributes definitions that will be needed by the remote nodes.
- The Maple user must explicitly start up the computation nodes using `Grid:-Setup("hpc")`. *Mathematica* starts these automatically when first required.
- The user does not have to state the number of nodes in the *Mathematica* version. This means that the *Mathematica* code will adapt to the environment, automatically knowing the number of CPUs available to it.
- The Maple programmer is responsible for ensuring that `Receive()` is called for each of the nodes (tracked using the counter **`i`** in the code). *Mathematica* does this automatically.
- In the Maple code, `Grid:-Setup("hpc")` is used to explicitly declare that the code should use remote hardware (as opposed to local processes on the same computer `Grid:-Setup("local")`). The *Mathematica* code will automatically distribute work to remote CPUs if it is aware of any, as well as to any local ones.
- *Mathematica* will automatically handle re-queueing of tasks to a new compute node should the node fail or become otherwise unavailable. There is no provision in the Maple example for node failure recovery.

Mathematica's rich automation allows the user to avoid most of the complexities of parallel programming, while Maple's bare-bones implementation of the basic concepts leaves much more of the work to the user.

Performance

The principle reason for parallelization is to improve speed.

Comparing the execution time of the two implementations reveals *Mathematica* to be a massive 30 times faster. Using this code, Maple would need a 128 CPU cluster to be able to match the performance of *Mathematica* running on a typical quad-core desktop computer.

Test performed using *Mathematica* 10.3 and Maple 2015.1 on a Windows 7 64-bit PC with Intel Core i7 3.07 GHz with 24GB RAM.
Timings are for parallel code execution only and exclude computation engine acquisition time.
Maple code was adjusted to use `Grid:-Setup("local")` and `numnodes=4`.
Timings averaged over five executions of the code were Maple: 5.312 seconds, *Mathematica*: 0.174 seconds.

Appendix: Code for approxint

Maple

The code for the simple Monte Carlo integrator used is as follows:

```
approxint := proc( expr, r::name=numeric..numeric,
                   { numsamples::integer := 1000 } )
    local f, randvals;

    f:= `if`( type(expr, procedure), expr, unapply(expr,lhs(r)) );

    randomize();
    randvals := LinearAlgebra:-RandomVector(numsamples,
        generator=evalf(rhs(r)), datatype=float[8]);
    (add( f(randvals[i]), i=1..numsamples )/numsamples) *
        (rhs(rhs(r)) - lhs(rhs(r)));
end proc:
```

Mathematica

The same function implemented in *Mathematica* is more compact:

```
approxInt[expr_, {var_Symbol, low_?NumericQ, high_?NumericQ}, n_: 1000] :=
 Mean[Function[{var}, expr] /@ RandomReal[{low, high}, n]] * (high - low)
```

For more comparison information, see
www.wolfram.com/mathematica/compare-mathematica/compare-mathematica-and-maple.html.

Maple® is a trademark of Maplesoft Inc.