Handle parallel (vectorized) objective functions in a new optimization wrapper package

Qiang Kou, Yann Richet

qkou@umail.iu.edu

January 9, 2014

1 Motivation

As we all konw, in some optimization methods, like Differential Evolution, the objective function $f(n)$ is evaluated many times. This project focuses on the challenges and benefits of parallel evaluations of $f(n)$. For example, in the simplex methods, when building a polytype, the points are passed to $f(n)$ sequentially. Much running time will be saved if the parallel evaluation of $f(x_1), f(x_2), f(x_3)$... is distributed on several cores/CPUs, especiall when $f(n)$ is quite CPU-expensive.

There are also similar situations in the numDeriv package, which is used to caculate the derivate of functions. In the caculation of numerical derivate, the $f(n)$ is also evaluated many times, so there is also an opportunity to speedup it by paralleled computing.

Thus, in many applications, we could expect a gain when calling $f(n)$ in a vectorized way, that is to say, for many evaluation points at the same call.

This project will deliver modifications of some optimization algorithms, to make the most of vectorized of objective functions. Moreover, when the objective functions is not preliminary vectorized, we can use a vectorization overlayer, as proposed in next section.

2 Methods to Implement

The foreach package provides a looping construct for executing R code repeatedly, which eases parallel execution, that is to say, it can execute for loops on multiple processors/cores on one computer, or even on multiple nodes of a cluster.

Besides, foreach package supports many different parallel backends, including openMPI and Redis.

	Differential Evolution Optimization/DEopt() on a dual-core CPU		
	no overhead	overhead 0.01	overhead 0.1
Original	0.315	92.635	905.999
doSNOW	25.794	75.791	483.715
doMC	13.219	60.608	467.627
doParallel	26.341	74.751	483.532

Table 1: Time for Differential Evolution Optimization/DEopt() on a dual-core CPU

So it provides such an easy way to parallalize for loops in R code, which are heavily used in optimization methods. Take the for loop in GAopt() function for example:

```
> for (s in snP)
+   vF[s] <- OF1(mP[, s])
```

By using the foreach package, it can run parallelly by modification like below:

```
> vF<-as.double(foreach(s=seq(snP)) %dopar% OF1(mP[, s]))
```

The optimization methods implemented in pure R language and in which the $f(n)$ is evaluated many times, are chosen to modify. The nmkb(), GAopt(), DEopt() and PSopt() are chosen to modify by these criteria. For a detailed review of optimization in R packages, please refer to the supplementary information.

3 Results

After modification, the 4 optimization functions are tested on Rosenbrock function. To simulate the CPU-expensive situation, some overhead is put on the function, so the advantage of parallel computing can be seen.

The detailed testing results using 3 different backends are list in 4 tables. Allthe test is run on a Thinkpad T61 dual-core laptop.

Besides the Rosenbrock function, other 5 test functions are chosen, including Ackley's function, Levy function, Powell function, Rastrigin function, and Schwefel function. They are all the classic test functions for optimization, and they are all high-dimensional, which are better for my test.

For the specific information on the test functions, please refer to the wikipedia page(http://en.wikipedia.org/wiki/Test_functions_for_optimization) and Dr. Bingham's website(http://www.sfu.ca/~ssurjano/optimization.html). And for the detailed testing results for each functions, please refer to the supplementary information.

	Nelder-Mead Algorithm/nmkb()			
	no overhead	overhead 0.01	overhead 0.1	overhead 1
Original	0.08	3.066	28.577	283.526
doSNOW	0.256	3.142	27.763	273.718
doMC	0.182	3.044	27.668	273.609
doParallel	0.258	3.187	27.763	273.704

Table 2: Time for Nelder-Mead Algorithm/nmkb() on a dual-core CPU

	Particle Swarm Optimization/PSopt()		
	no overhead	overhead 0.01	overhead 0.1
Original	0.238	61.849	604.32
doSNOW	21.943	55.766	328.254
doMC	10.403	43.486	314.517
doParallel	22.515	55.059	327.684

Table 3: Time for Particle Swarm Optimization/PSopt() on a dual-core CPU

	Genetic Algorithm/GAopt()		
	no overhead	overhead 0.01	overhead 0.1
Original	0.138	20.915	202.855
doSNOW	3.12	14.474	105.787
doMC	4.225	14.776	105.6
doParallel	3.061	14.347	105.692

Table 4: Time for Genetic Algorithm/GAopt() on a dual-core CPU

	$sin(x) + cos(x)$			
	no overhead	overhead 0.01	overhead 0.1	overhead 1
Original	0.004	0.038	0.308	3.007
doMC	0.038	0.053	0.256	2.047

Table 5: Test of jacobian() on $sin(x) + cos(x)$ on a dual-core CPU

For the numDeriv package, there are only 4 functions, hessian(to build Hessian matrix), jacobian(to build Jacobian matrix), grad(to calculate the derivative) and genD(to build Bates matrix).

The hessian function is based on the other three and not much work can be done. The grad and jacobian are modified using foreach, the testing result can be found in Table 5.

The genD() hasn't been modified since parallel computing will not have advantage on this function. There will be more discussion on it.

4 Discussion

From the testing results, we can see time gain by parallel evaluation. And the 3 different backends on my laptop make no much difference.

However, if the number of possible parralel evaluations is not enough high, the gain will be very thin, even negative. This is typically can be seen in Nelder Mead/nmkb() results.

Take the GAopt() for example, in the loop below, the number of $f(n)$ evaluation is the generation number for genetic algorithms, and it is 20 at least. So the gain in running time can be seen when $f(n)$ is more or less CPU-expensive.

```
> vF<-as.double(foreach(s=seq(snP)) %dopar% OF1(mP[, s]))
```

However, in the loop of genD(), $r = 4$ happens in most situations, so no gain in running time can be seen. So the modification will be meaningless in most situations, since messaging between cores or nodes will consume too much time.

```
> for(k in 1:r)
+ {
+         f1 <- func(x+(i==(1:p))*h, ...)
+         f2 <- func(x-(i==(1:p))*h, ...)
+         Daprox[,k] <- (f1 - f2)  / (2*h[i])
+         Haprox[,k] <- (f1-2*f0+f2)/ h[i]^2
+         h <- h/v
+           NULL
+ }
```

So we put some more arguments in the modified functions like below.

```
> DEopt.vectorized(OF, algo = list(), vectorized = FALSE, foreach.option = list(methods = "doMC", nod
>
```

There are choices for the argument “vectorized”, “FALSE”, “apply” or “foreach”. So you can choose not vectorize the objective function, or vectorize through “apply” or “foreach”, facing different objective functions. If “foreach” is chosen, more options should be specified, including the backend and node number.