研究課題 グリッドコンピューティング環境の構築と評価
種別 指定研究
代表者 武田俊之(情報メディア教育センター)
研究員 地道正行(商学部)、岡田孝(理工学部)

1. 概要

上ヶ原グリッドシステム主に文系の研究分野をターゲットとしている。したがって、一般的なハイ・パフォーマンス・コンピューティング(HPS)より容易に処理プログラムを記述することが要求される。本プロジェクトでは、実際の研究課題のプログラムを記述しながら、以下の計算環境構築をおこなった。

1. GIobus Toolkit [1] を利用して SAS、R など統計パッケージを並列に実行し、結果を集計することができる環境にすること。
2. MPI (Message Passing Interface) [2] を利用したプログラムが作成可能な環境にすること。

MPI は理解が容易なライブラリ・インターフェースで並列プログラミングにおいては広く用いられている。本研究の目的は、R、Python などのスクリプト言語から MPI を利用するライブラリを導入してさらに容易にグリッドシステムを利用することを可能にすることである。

プログラミング環境の構築は主に武田研究員がおこない、グリッド・アプリケーションとして地道研究員による「実行可能型一般化リッジ回帰推定量の正確なモーメントの数値評価」と岡田研究員による「GRID 技術によるカスケードモデルの高速化」を実施した。

2. MPI プログラミング環境の構築

MPI は C 言語や Fortran から呼び出すことのできる並列プログラミングのためのライブラリの仕様である。このライブラリの基本はメッセージ・パッシングによって並列処理を実現するよう設計されている。MPI の実装としては、MPICH や LAM/MPI が有名である。MPI はさまざまなプログラム環境から利用可能であるが、C 言語のように煩雑なプログラミングが必要となると利用する研究者にとっては不便を強いることになる。ゆえに、統計システム R および、スクリプト言語 Python から MPI を利用する環境を構築、実際の研究に適応することを試みた。

2.1. マスター・ワーカー・モデル

マスター・ワーカー・モデルは、並列プログラミングをもっとも容易かつ柔軟に実現するものの一つである。マスター・ワーカーにおいて、マスター・プロセスは並列実行されるプロセスをコントロールするために、ワーカー・プロセスの集合を生成する。ワーカーはマスターから受けとったタスクを実行して、結果をマスターに返す。マスターは集まった結果から最終的な結果を計算する。

マスター・ワーカーの主な実装としては、プッシュ型とプル型があげられる。プッシュ型はマスター・プロセスからタスクを送信する場合で、プル型はワーカー・プロセスがタスクを要求する。

今回は、ワーカーの性能にばらつきがある場合には、より効率的であると思われるプル型のライブラリを構築した。

マスター・ワーカー・モデルを MPI 上に実装した Rmpi と Pypar を利用することに決めた。

2.2. Rmpi

Rmpi は R から利用可能な MPI へのインターフェース(ラッパ)である。Rmpi は LAM／MPI の上で実装されている。Rmpi はワーカー・プロセスを MPI のノード上に起動する。このワーカー・プロセスは命令が送られてくるまでウエイトしている。

他の MPl アプリケーションと異なり、R のプログラムを mpirun で起動する必要はない。スレーブ・プロセスは R の命令に従って起動される。

主な Rmpi のコマンドは以下のとおりである。

- 1ibrary(”Rmpi”)
- mpi.spawn.Rslaves([nslaves＝＃])
- mpi.close.Rslaves()
- mpi.quit([saving=yes/no])
- mpi.comm.size()
- mpi.comm.rank()
- mpi.bcast.Rob2slave(object)
- mpi.send.Robj(object, destination, tag)
- mpi.recv.Ro 叫(mpi.any.source(), mPi.any.tag())
- mpi.get.sourcetag()
- mpi.bcast.cmd(“R code”)
- mpi.remote.exec(“R code”)

武田研究員が Rmpi で実装したマスター・ワーカーのライブラリは以下の手順で実行される。

1. マスター・プロセスは、ライブラリ Rmpi をロードして、ワーカー・プロセスを起動する。
2. ワーカーの実行する関数、do_task を定義する。
3. マスターが実行する関数、do_with_results を定義する。
4. 必要なら、do_task や do_with_results で使用する関数を定義する。
5. タスクを生成する。
6. ワーカーに、do_task およびそこで使用される関数、変数を転送する。
   mpi.bcast.Robj2slave(do_task)
   mpi.bcast.Robj2slave(worker.main)
7. ワーカーの関数を起動する。
   mpi.bcast.cmd(worker.main())
8. マスターのメイン関数を実行する。
   master.main()
9. 各ワーカーで計算された結果はマスターに転送され、do_with_result で処理される。もし、すべての計算が終わった後で、計算結果を利用した処理が必要ならおこなう。

2.3. Pypar

Pypar［2］は MPI の Python バインディングの一つで、もっともインストールが容易でありながら、以下の優れた特徴を持つ。(1) どんな型のオブジェクトでも送信することができる。(2) プログラミングが容易である。(3) 効率的である。(4) 軽量である。(5) 拡張が容易である。

Pypar の例として、台形公式のプログラムを次ページに掲載する。

2.4. 参考文献

[1] http://www.globus.org/

[2] http://www-unix.mcs.anl.gov/mpi/

[3] http://www.stats.uwo.ca/faculty/yu/Rmpi

[4] http://datamining.anu.edu.au/~ole/pypar

```
#! /usr/bin/env python

from math import sqrt
import sys

try:
        import pypar
except:
    raise 'Module pypar must be present to run'

def f(x):
    return sqrt(1-x**2)
def trap(a, b, n, h):
    integral = (f(a) + f(b)) / 2
    x = a
    for i in range(1, n-1):
        x = x + h
        integral = integral + f(x)
    integral = integral * h
    return integral

def main()
    a = 0.0
    b = 1.0
    n = 1200
    myid = pypar.rank()
    numproc = pypar.size()
    node = pypar.get_processor_name()

    h = (b - a) / n
    local_n = n / numproc
    local_a = a + myid * local_n * h
    local_b = local_a + local_n * h
    integral = trap(local_a, local_b, local_n, h)

    if numproc < 2:
        sys.stderr.write("This program must run on at least 2 processors to continue (numproc = %d)"
                         % numproc)
        pypar.abort()
    if myid == 0:
        sys.stderr.write("a = %f b = %f n = %d numproc = %d¥n" % (a, b, n, numproc))
    if myid == 0:
        total = integral
        for source in range(1, numproc):
            integral, status = pypar.receive(source, return_status=True)
            sys.stderr.write("[Master]: received result %f from node '%d'¥n" %(integral, status.source))
            total = total + integral
    else:
        pypar.send(integral, 0)
        sys.stderr.write("[Worker %d]: sent result %f to node '%d'¥n" % (myid, integral, 0))
    pypar.finalize()
    if myid == 0:
        sys.stderr.write("With n = %d trapezoids, our estimate" % n)
        sys.stderr.write("of the integral from %f to %f = %f¥n" % (a, b, total))

if __name__ == '__main__':
    main()
```

3. 実行可能型一般化リッジ回帰推定量の正確なモーメントの数値評価(地道)

線形回帰モデルにおける説明変数行列の列ベクトル間に多重共線性(multicollinearity) が存在するときに、回帰係数ベクトルに対する通常最小自乗(Ordinary Least Square; OLS) 推定量の分散が発散することから、推定精度に関する問題が発生する可能性があることが指摘されてきた。 この問題に対して、OLS推定量のかわりに、一般化リッジ回帰(Generalized Ridge Regression; GRR)推定量がHoerl & Kennard (1970)によって提案された。このGRR 推定量を実際に使用するときは、ある種の係数(リッジ係数とよばれる。)を決定する必要があり、いくつかのアルゴリズムが提案されている。この決定されたリッジ係数をGRR推定量の適切な箇所にプラグ・インしたものは、実行可能型一般化リッジ回帰(Feasible GRR; FGRR) 推定量とよばれる。ここで、FGRR 推定量の良さを測る基準である平均2 乗誤差(Mean Squared Error; MSE)などを求める際に、この推定量のモーメントが必要となるけれども、その正確なモーメントを求めることは困難であるという問題がある。これは、リッジ係数を決定するためのアルゴリズムがいくつかの確率変数に依存することによる。この種の問題を扱った研究はそれほど多くなく、 Dwivedi、 Srivastava & Hall (1980)とSrivastava & Chaturvedi (1983) が先駆的な仕事といえる。本研究では、 FGRR 推定量の正確なモーメントを数値的に求めることを目的とした。とくにクロスモーメントの値を具体的に求めることは単一の計算機環境では困難であったため、収束性などをグリッドコンピューティング環境を用いることによって数値的に評価した。なお、この結果はJimichi (2005) にまとめられた。ただし、ここで与えられた結果は比較的小さなパラメータの値に対するものであり、現時点での評価は限定的なものである[1] 。よって、今後さらに多様なパラメータの場合に対して評価を行う予定である。今回の計算にはグリッドシステム(Globus Toolkit 3.2) 上のプログラミング環境であるLAM/MPI、Rmpi そしてR を利用した。

3.1. 参考文献

[1] Dwivedi, T. D., Srivastava, V. K. and Hall, R. L. (1980). Finite sample properties of ridge estimators. Technometrics 22, 205–212.

[2] Hoerl, A. E. and Kennard, R. W. (1970). Ridge regression: biased estimation for nonorthogonal problems. Technometrics 12, 55–67.

[3] Jimichi, M. (2005) Exact moments of feasible generalized ridge regression estimator and numerical evaluations, Submitted.

[4] Srivastava, V. K. and Chaturvedi, A. (1983). Some properties of the distribution of an operational ridge estimator. Metrika 30, 227–237.

1

[1] このことは級数の収束性に依存していることによる。