グリッドコンピューティング

Grid パッケージには、マルチプロセス並列処理実行のための手法が含まれています。Maple 2015 には、同一マシンによる無制限の並列処理実行機能が内蔵されています。追加のツールボックスやライセンスを取得することなく、必要なだけ並列処理をスポーンできます。

Maple 2015 では、MPI のようなメッセージ通信プロトコルを取り除く新しいコマンドを使用して、並列ジョブをより簡単に開始できます。結果として、非常にシンプルで直感的なインターフェースから、コマンドを実行し、データを処理できるようになりました。

新規コマンド：Run、Set、Get、GetLastResult、Wait、WaitForFirst

```
with(Grid);
```

[*Barrier*, *Get*, *GetLastResult*, *Interrupt*, *Launch*, *Map*, *MyNode*, *NumNodes*, *Receive*, *Run*, *Send*, *Seq*, *Server*, *Set*, *Setup*, *Status*, *Wait*, *WaitForFirst*] **(1.1**

主要な新規コマンド Grid:-Run は、個々のノード上で非同期ジョブのスポーンを可能にします。これは、以下を意味します。

- 結果を待つ必要がない
- 1 つ以上のノードで使用可能
- すべてのノードが使用中の場合、Run は暗黙的に待機し、個々のノードの結果を返す
- 通常は Send および Receive を必要としない

例：バックグラウンドジョブ

この例では、2 つのジョブをバックグラウンドで実行する方法について説明します。assignto オプションにより、ローカルセッションの結果を取り込むことができます。同一ノード上で結果を使用する前や新規ジョブを開始する前に、必ず現在のジョブが完了するのを待ってください。

```
> Grid:-Run(0, Optimization:-NLPSolve, [sin(x)/x, x = 1 ..30], 'assignto' = 'ans0')
```

```
> Grid:-Run(1, Optimization:-NLPSolve, [x^3 + 2 x y - 2 y^2, x = -10..10, y = -10..10, initialpoint = {x = 3, y = 4}, 'maximize'], 'assignto' =
      'ans1')
> Grid:-Wait( )
> ans0
```

$$[-0.0424796169776126, [x = 23.5194525023235]] \quad (1.1.1$$

```
> ans1
```

$$[1050., [x = 10., y = 5.]] \quad (1.1.2$$

これらのコマンドは、後ほど示すように、逐次実行することも可能です。上記のとおり 2 つの計算で Run コマンドを使用することで、両方のコマンドを同時に実行できます。最初のコマンドは、2 番目のコマンドを開始する前に終了させておく必要はありません。2 つのコマンドを並列で実行することで、実行時間を半減させることができます。多数のジョブを同時に実行することで、逐次実行するジョブの場合と比較してより大きなパフォーマンスの改善を実現できます。

```
> Optimization:-NLPSolve( sin(x)/x, x = 1..30 );
```

$$[-0.0424796169776126, [x = 23.5194525023235]] \quad (1.1.3$$

```
> Optimization:-NLPSolve( x^3 + 2 x y - 2 y^2, x = -10..10, y = -10..10, initialpoint = {x = 3, y = 4},'maximize' );
```

$$[1050., [x = 10., y = 5.]] \quad (1.1.4$$

例：すべてのノードからの結果

この例では、複数ノードによる計算の結果が、どのように配列に戻されるかを説明します。Grid:-Set コマンドは、現在のセッションからの値をワーカーノードに割り当てるために使用されることに注意してください。この場合、work プロシージャはローカルで定義されるため、Set コマンドが適用されるまで、並列処理プロセスに対しては不明のままとなります。

```
> work := proc( )
      randomize( );
      cat("node ", Grid:-MyNode( ), " computed ", rand(1..10)( ) );
```

```
    end:
> Grid:-Set(work);
> R := Grid:-Run(work)
      R := Array(0 ..3, {0 = "node 0 computed 9", 1 = "node 1 computed 10", 2 = "node 2 computed 5", 3 = "node 3 computed 8"})   (1.2.1
> R[0];
                                        "node 0 computed 9"                                                                  (1.2.2
> R[1];
                                       "node 1 computed 10"                                                                  (1.2.3
> R[2];
                                        "node 2 computed 5"                                                                  (1.2.4
> R[3];
                                        "node 3 computed 8"                                                                  (1.2.5
```

Run に対する最初のパラメータに指定されたターゲットノードがない場合、4 コアマシンは自動的に 4 つのジョブをスポーンします。すべての結果は配列に戻され、結果は対応するノード番号に記されます。

例：コンペティション - 最初に終了するジョブ優先

この例では、4 つの異なるジョブを実行し、一番最初に終了するジョブのみを待ちます。その後、ほかのジョブを終了します。先ほど説明したとおり、リモートノード上で delay プロシージャを定義する Grid:-Set の使用に注意してください。

```
> method0 := proc( )
    Threads:-Sleep(15) :
    "result of method 0";
  end:
  Grid:-Set(0, method0);
> method1 := proc( )
    Threads:-Sleep(13) :
    "result of method 1";
```

```
    end:
    Grid:-Set(1, method1);
> method2 := proc( )
      Threads:-Sleep(3) :
      "result of method 2";
    end:
    Grid:-Set(2, method2);
> method3 := proc( )
      Threads:-Sleep(9) :
      "result of method 3";
    end:
    Grid:-Set(3, method3);
> Grid:-Run(0, method0);
  Grid:-Run(1, method1);
  Grid:-Run(2, method2);
  Grid:-Run(3, method3);
> n := Grid:-WaitForFirst( )
```

$$n := 2 \qquad \text{(1.3.1}$$

```
> Grid:-Interrupt( )
> Grid:-Wait( )
> Grid:-GetLastResult(n)
```

$$\text{"result of method 2"} \qquad \text{(1.3.2}$$

Grid:-WaitForFirst コマンドは、計算を終了させるために最初のノードのノード番号を返します。次に、Grid:-GetLastResult を使用して、そのノードで最後に計算された値を取得します。計算された結果が名前を持つ場合、その値を取得するために Get コマンドを使用することもできます。

Grid:-Map および Grid:-Seq の改良点

Grid:-Map および Grid:-Seq が、Grid:-Run を使用して再実装されました。これは、以下を意味します。

- スポーン実行プロセスのオーバーヘッドの低減
- より適切な作業の分配
- オプションの tasksize パラメータ

tasksize

この例では、作業が不均等に区分されています。リスト末尾にある計算量の大きな計算の式列は、デフォルトの tasksize の調整が必要な場合があります。

```
> data := [seq(5^min(i, 10), i = 1 ..12)];
```

$$data := [5, 25, 125, 625, 3125, 15625, 78125, 390625, 1953125, 9765625, 9765625, 9765625] \quad (2.1.1)$$

```
> time[real]( Grid:-Map(proc(n) printf("node %d computing %d\n", Grid:-MyNode( ), n); add(i, i = 1 ..n); end proc, data) );
```

```
node 2 computing 78125
node 1 computing 625
node 3 computing 9765625
node 0 computing 5
node 0 computing 25
node 1 computing 3125
node 0 computing 125
node 1 computing 15625
node 2 computing 390625
node 2 computing 1953125
node 3 computing 9765625
node 3 computing 9765625
```

$$2.312 \quad (2.1.2)$$

計算には少量の数字しか含まれないため、デフォルトではオペレーションを均等な区分に分割します。ノード 0 が最初の 3 つのデータ点を取得し、ノード 1 が次の 3 つを取得する、という形で分割します。この場合、仕事が不均等に配分されているため、最終的にノード 3 が最も大きな 3 つのケースを計算することになり、ここがボトルネックになります。

この種の例でタイミングを最適化するには、tasksize をより小さな値に設定します。tasksize=1 に設定すると、アルゴリズムによってデー

タが 1 要素 (前例の 3 要素ではなく) ずつに分割されます。12 のタスクが存在しますが、計算ノードは 4 つのみとなります。各ノードは 1 つのタスクを終了してから次のタスクを要求します。

```
> time[real]( Grid:-Map[tasksize = 1](proc(n) printf("node %d computing %d\n", Grid:-MyNode( ), n); add(i, i = 1 ..n); end proc,
      data) );
node 0 computing 5
node 1 computing 25
node 3 computing 625
node 2 computing 125
node 0 computing 3125
node 1 computing 15625
node 2 computing 78125
node 3 computing 390625
node 0 computing 1953125
node 1 computing 9765625
node 2 computing 9765625
node 3 computing 9765625
                                  1.609
```

(2.1.3

time[current]

time() コマンドの新しいオプションにより、現在時刻の1 秒以下の詳細な測定が可能になります。ほかのプロセスで実行を比較する場合に有用です。

この例では、すべてのノードにおけるジョブの実行前、実行中、および実行後の現在時刻を含むメッセージを表示します。

```
> show_time := proc( msg := "" )
    local t := time[current]();
    printf("node=%d current-time=%s.%d: %s\n",
      Grid:-MyNode(),
      StringTools:-FormatTime("%I:%M:%S",timestamp=trunc(t)),
        trunc(10^6*(t - trunc(t))),
      msg);
  end proc:
```

```
> Grid:-Set(show_time)

> show_time("initialize")
  Grid:-Run(show_time, ["worker"], 'wait' = true) :
  show_time("done")
```

```
node=0 current-time=02:32:11.781000: initialize
node=0 current-time=02:32:11.781000: worker
node=1 current-time=02:32:11.781000: worker
node=2 current-time=02:32:11.781000: worker
node=3 current-time=02:32:11.797000: worker
node=0 current-time=02:32:11.797000: done
```