ジョブ投入と待ち合わせの出来るジョブ制御スクリプト：オーガニックジョブコントローラの試作

上田 晴康*　吉田 武俊†　安里 彰†

グリッド技術により、大量のノードに大量のジョブを投入することが出来るようになった。しかし大量のジョブを管理するのは難しく、ジョブ数の増大とともに増える工数が大きな問題であった。またジョブのワークフローを記述したいというニーズがあったが、大量のジョブ投入の場合困難であった。そこで、ジョブを制御するための専用スクリプトとその処理系からなるオーガニックジョブコントロール技術を開発した。これにより、ジョブ管理が容易になったほか、先行ジョブの結果に基づく後続ジョブの動的生成や、部分的なジョブの再実行など、複雑なジョブ制御が可能になった。この技術を大量シミュレーションジョブの投入に適用した結果、ユーザ作業が 1/4 になる効果が示された。

Organic Job Controller — A script language to describe submission, synchronization and dependency of jobs —

Haruyasu Ueda*　Taketoshi Yoshida†　Akira Asato†

To make full use of grid system, users submit a large amount of jobs. With a current job queuing system, it is difficult to check completion, delete or re-submit such jobs without script programs. And, to describe workflow for many jobs is almost impossible. To solve them, we developed Organic Job Controler with a new designed script and its interpreter. This technology enables user to manage many jobs as well as to control jobs, e.g., dynamic job submission based on the result from precedent jobs or partial job deletion without writing scripts. The result of a experiment shows the man-hour to manage jobs is dramatically reduced.

1　はじめに

グリッド関連技術 [1] により、非常に多くのノードの結合された計算環境が容易に構築できるようになってきた。グリッド環境では計算機性能のばらつきが大きいため、並列プログラムを開発するのは困難であり、逐次プログラムをバッチジョブとして大量に実行して計算資源を利用する機会が増えると考えられる。例えば、200 個のノードの結合されたサーバに、1 時間かかるジョブを投入して、3 日の間休みなく働かせるとした場合、一度に、3*24*200=14400 個のジョブを投入することになる。

従来のバッチジョブ投入システム [10, 2, 4] は、あらかじめジョブスクリプトファイルを用意し、一つずつジョブを投入する必要があり、ジョブの終了確認はシステムが返すジョブ ID に基づいてジョブの終了状態を問い合わせる必要があった。このため、非常に多くのジョブを投入する場合、これらの作業を行うために別途 sh,perl,make などのスクリプト言語でプログラムを作成する必要がある。また、ジョブを一度投入しただけで完全なデータが取れることはまれで、一部のジョブをやり直したり、類似のジョブを追加したりする作業がしばしば発生するので、そのたびに作成するスクリプトや実行ログの管理の手間も問題となる。

また、多くのジョブを投入する際に有用な方法としてパラメータスウィープがあるが、ジョブの終了を待って結果を取りまとめたり、ジョブの結果を用いて追加のジョブを投入したりする複雑なワークフローとパラメータスィープを同時に記述するのは従

*富士通 (株) FUJITSU LIMITED
†(株) 富士通研究所 FUJITSU LABS. LIMITED
{ueda,tyoshida,asato}@labs.fujitsu.com

来は非常に困難であった。

これらの問題を解決するために、大量のジョブの管理やワークフロー、パラメータスウィープといった機能を持ったユーザ支援環境が必要となってくる。

そこで、我々は大量のジョブを依存関係を含めて定義する専用スクリプト、スクリプトを解釈して名前付きのジョブを投入するインタプリタ、ジョブをユーザのつけた名前で管理する管理部から構成されるオーガニックジョブコントロール技術を開発した [15]。本技術によって、ジョブ管理が容易に行えるようになったことに加え、先行ジョブの結果に基づく後続ジョブの動的生成や、部分的なジョブの再実行など、インテリジェントなジョブ実行制御が可能になったので、本技術の内容について報告する。また、この技術を実際の業務に対して提供し評価を行ったので、その評価結果についても報告する。

2 類似研究

globus[1], UNICORE[12], Ninf[3] などのグリッドミドルウェアに対して利用しやすいインターフェースを提供する GCE(Grid Computing Environment)[13] の研究が盛んに行われている。その中でも大量にジョブ投入する枠組、例えば task farming[8], パラメータスウィープ [9] による大量ジョブ投入に対するユーザ環境 [11] やワークフローを記述するための枠組がいくつか提案されている。

nimrod,nimrod/G[5, 7, 6] はパラメータスィープに関して初期からあるシステムである。ユーザの記述したパラメータファイルを元に、ワークステーションクラスタや globus 上でジョブを実行出来る。しかし、ワークフローに準じた処理は一つずつのジョブが終了した時と全てのジョブが終了した時の処理を記述できるだけで、いくつかのジョブを待つことは出来ない。また部分的なジョブの再実行をするにはパラメータファイルを編集する必要がある。

APST(AppLeS Parameter Sweep Template)[9] は、パラメータスウィープを目的としているが、より柔軟なスケジューリングをすることを目的に開発された。パラメータスィープに特化した GUI を持っているが、ワークフロー的な処理はグラフを動的に表示する事のみに限られている。

ワークフローが指定できる環境としては、UNICORE[12] がある。ユーザは GUI からワークフローを指定することが出来るが、パラメータスウィープ等でジョブ数が多いと GUI でそれらを一括して指定するのは困難である。

SCIRun[14] は、ユーザが GUI で指定したワークフローに従って、リソース計算機にあらかじめインストールされた MATLAB など HPC 用ライブラリを呼び出して計算することが出来る。しかし、SCIRun は統合環境であるがゆえに、パラメータスィープに対応したライブラリがある場合に限ってパラメータスウィープを行え、またワークフローもライブラリの単位で行えるのみで汎用性に欠ける。

ILab[17, 11] は、パラメータスウィープを主目的としていながら、条件を指定して条件が満たされるまでループするなどワークフローのような記述が出来る。しかし、この記述をするためには ILab の perl ライブラリを意識して待ち合わせを手続き的に記述する必要があり、利用が難しい。

いずれの研究においても大量のジョブ投入ができかつジョブ間のワークフローを簡単に宣言的に記述する方法は提供されていない。このため、大量のジョブ投入とその結果の取りまとめなどを考えた場合、単純なパラメータスウィープでジョブを投入し、取りまとめなど後で実行すべきジョブは人手を介して投入するしかなかった。

3 オーガニックジョブコントローラ

オーガニックジョブコントローラ (以下 OJC) はワークフローやパラメータスウィープを簡単に実現できる枠組みであり、以下のような特徴をもつ。

- パラメータスウィープを含む task farming を容易に行える。ジョブ投入は perl あるいは sh スクリプトのループ中から行うことができ、投入指示に記述したジョブの雛型に関して変数の展開を行うことが出来る。
- ジョブに名前を付けることが出来、ジョブの終了確認を簡単に行うことが出来る。
- ワークフローは待つべきジョブの名前に関するパターンマッチとして宣言的に記述でき、強力な待ち合わせができる。

3.1 アーキテクチャ

OJC は、大きく分けて名前付きのジョブを重複しないように投入する名前付きジョブ管理部とジョ

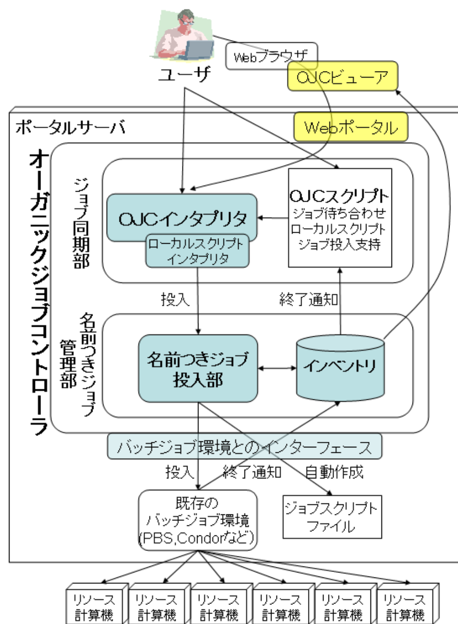

図 1: OJC のアーキテクチャ

ブの終了待ちをして適当なタイミングでジョブ投入を行うジョブ同期部の二つの部分からなる。OJC は外部とのインターフェースとして、ユーザとのインターフェース (WWW ポータル経由またはコマンドライン経由) と、既存のバッチジョブ環境とのインターフェースを持つ (図 1)。

OJC は OJC ビューアを除いて全て perl 言語で実装されている。OJC ビューアは JAVA applet として実装されている。

このアーキテクチャでジョブを処理するまでの手順は以下のようになる。

• ユーザは、あらかじめ OJC スクリプトと実行すべきバイナリ、入力ファイルを用意しておき、ftp や WWW ポータルなどでポータルサーバに転送する。

OJC スクリプトには、ジョブの待ち合わせ情報、ローカルスクリプトと呼ばれる待ち合わせ後に実行する小さなスクリプト、およびローカルスクリプト中でのジョブ投入指示を記述しておく。ローカルスクリプトは既存のスクリプト言語 (sh または perl) で記述する。

• ユーザが OJC インタプリタの起動を行う。OJC インタプリタからローカルスクリプトのインタプリタが起動される。

• OJC インタプリタは OJC スクリプトの記述に従って、ジョブの待ち合わせを行い、適切なタイミングでローカルスクリプトをローカルスクリプトインタプリタに送って実行する。

• ローカルスクリプトインタプリタは普通にスクリプトの実行処理を行い、ジョブ投入指示を見つけると、ジョブの名前とともに名前付きジョブ投入部を呼び出す。

• 名前付きジョブ投入部は、まず同じ名前のジョブが過去に投入済みかインベントリに問い合わせをし、未投入の場合に限ってジョブスクリプトファイルを自動的に作成し、ジョブ投入を行う。ジョブ投入が成功すると、バッチジョブ環境が返したジョブ ID をインベントリに保存し、投入済みであることを記録する。

• インベントリは、ユーザが投入したジョブに関してジョブの名前とジョブの実行状況などのデータを管理する。実装上は一つのディレクトリであり、ジョブの情報はそれぞれ一つのファイルにテキストファイルで収められている。インベントリの内容は、ジョブ投入時、ジョブキューから出て実行を開始した時、およびジョブの完了時に情報が更新される。

• バッチジョブ環境は、投入されたジョブをキューに入れて管理し、空いているリソース計算機があれば、そこで実行させる。

バッチジョブ環境とのインターフェース部分は、ジョブ投入の qsub、ジョブ確認の qstat、ジョブ削除の qdel だけが必要であり、OpenPBS であればインターフェース部分なしに直接利用できる。

• OJC インタプリタはインベントリを監視しており、全てのジョブが完了するまで実行し続ける。

• 実行中のジョブの様子はコマンドラインから確認できるほか、OJC ビューアを用いることで、インベントリ中のジョブの情報を見ることが出来る。必要があれば、ビューアからもインタプリタを止めたりジョブの入出力ファイルを見ることが出来る。

3.2 OJC スクリプト

OJC スクリプトは、ジョブの待ち合わせ、投入などを行う OJC 独自のシンタックスと既存のスクリプト言語を利用して制御などを行う部分からなる。OJC に特有の部分は`{{ }}` で囲まれていて既存のスクリプト言語と混乱しないようになっている。OJC スクリプトの例を図 2 に示す。

3.2.1 top ブロックと when ブロック

OJC スクリプトは、`top {{`**ローカルスクリプト** `}}` (top ブロック) または `when {{`**ジョブ名**`...}}{{`

```
top {{
  for $paramA (3, 5, 7, 11, 13) {
    for ($paramB=0; $paramB<10;$paramB++){
      $ENV{paramA} = $paramA;
      $ENV{paramB} = $paramB;
      do_job {{ paramA=$paramA
                paramB=$paramB }} {{
         system("test_bin $paramA $paramB
                > result.$paramA.$paramB");
      }}
    }
  }
}}
when {{ paramA=$paramA paramB=* }} {{
    system("cat result.$paramA.*
             > result_A.$paramA");
}}
```

図 2: OJC スクリプトの例 (ローカルスクリプト言語は perl)

ローカルスクリプト}} (when ブロック) が並んだものとして記述される。これらは、ローカルスクリプトの実行のタイミングを記述するもので、top ブロックは実行開始時に、when ブロックは指定されたジョブが全て終了した後に実行することを意味する。when ブロックは一種のバリア同期である。

ローカルスクリプトは既存のスクリプト言語で記述する。ローカルスクリプトは、実行すべきタイミングになると、順にローカルスクリプトインタプリタに送られて逐次に実行される。複数のローカルスクリプトが同時に実行されることはない。ローカルスクリプト間で情報を共有する際には、大域的な変数を利用することが出来る。

ローカルスクリプトの中にはジョブ投入を指示するために do_job **ジョブ名**{{**ジョブスクリプト** }} という特別なブロックを記述することが出来る。ジョブスクリプトには、リソース計算機で実行すべき内容を記述する。ジョブスクリプトファイル中で参照したい変数は、ローカルスクリプトで環境変数に設定しておく。

名前付きジョブ投入部は、ジョブ投入に際して、ジョブスクリプトファイルを作成する。このファイルは、OJC に記述されたジョブスクリプトを雛型として、投入時点の環境変数の再構成などジョブ実行時に必要なスクリプトを付加して作成される。

3.2.2 ジョブ名と待ち合わせパターン

ジョブ名は {{**名前**=**値** ...}} の形式で指定する。パラメータスウィープを実行しやすいよう、ジョブ名には複数のパラメータ名を組み合わせて指定できる。ジョブ名の指定には、ローカルスクリプトの変数を参照でき、名前付きジョブ投入部の呼び出し時に展開される。

when ブロックの待つべきジョブの指定にはジョブ名を表すパターンを記述することが出来る。ジョブ名のパターンとしては、 * や ? のワイルドカード文字と$**変数** というパターン変数を記述することが出来る。ワイルドカード文字は、ジョブ名を文字列とみなしてマッチするジョブのみ待ち合わせる。パターン変数は、1 文字以上の文字列とマッチするが、特別な効果としてマッチした文字列を共有するジョブ群をグルーピングしジョブ群が終了した時にローカルスクリプトを実行するという効果をもつ。

例えば、図 2 の例では、{{ paramA=$paramA paramB=* }} というジョブ名のパターンが記述されているが、これは、paramA が同じ値を持つ任意の paramB を持つジョブをグルーピングする。この例では top ブロックの中で paramA が 3,5,7,9,11 の 5 種、paramB が 10 種で合計 50 のジョブが投入されるため、when ブロックのローカルスクリプトは paramA=3 のジョブ 10 個が終了したときに 1 回、paramA=5 のジョブ 10 個が終了したときに 1 回.. のように合計 5 回実行される。そして、それぞれの paramA に関する結果の集計を行う。

3.3 OJC の高度な利用法

OJC は単なるジョブ管理とパラメータスィープが出来るだけでなく以下のような高度の利用方法が可能である。

3.3.1 不定個のジョブの待ち合わせ

不定個のジョブの終了を待つ場合の例として、並列ソートがある。並列ソートは、入力ファイルを一定のサイズ毎に分割して、それぞれのファイルでソートを行った後マージを行う。ジョブの数は入力ファイルのサイズに依存して決まるので、待つジョブの数を静的に記述するのはとても難しい。通常のプログラム言語を使った場合、待つべきジョブのリ

```
top {{
  for $file ('fileA', 'fileB', 'fileC') {
    $jobs = 0;
    $line = 0;
    open(F, $file);
    open(O, "> $file.$jobs");
    while (<F>) {
      print O $_;
      $line ++;
      if ($line > 1000000) {
        $ENV{file} = $file;
        $ENV{jobs} = $jobs;
        do_job {{ file = $file
                  part = $jobs }} {{
          system('sort',
                 '-o', "$file.$jobs.out",
                 "$file.$jobs");
        }}
        $jobs++;
        $line = 0;
        open(O, "> $file.$jobs");
      }
    }
    $ENV{file} = $file;
    $ENV{jobs} = $jobs;
    do_job {{ file = $file
              part = $jobs }} {{
      system('sort',
             '-o', "$file.$jobs.out",
             "$file.$jobs");
    }}
  }
}}
when {{ file=$file part=* }} {{
  do_job {{ file = $file summary=y}} {{
    system("sort -m $file.*.out
             -o $file.out");
  }}
}}
```

図 3: 不定個のジョブ投入を行う例
3 個のファイルの並列ソート

ストといった変数を用いて手続き的に待ち合わせを書かなくてはならない。

OJC では、それらはジョブのパターンの待ち合わせとして宣言的に書くことが出来る。また、複数の並列ソートを行うような場合でもジョブのグルーピング機能があるため、一つの when ブロックの記述ですむ (図 3)。

3.3.2 動的ジョブ生成

動的にジョブの数が増えるような例として、ジョブの実行結果を元に最適化を行うような場合がある。この場合、それぞれのジョブの終了を待ってから、ジョブの出力を取り出し、その出力に基づいてパラメータを最適化しつつ新たなジョブを投入するという作業が必要である。OJC を用いれば、以下のような二つの when ブロックを記述するだけよい (図 4)。

- 各ジョブの終了を待って、出力結果を最適化アルゴリズムに入力し、最適化アルゴリズムが出力したパラメータで新たなジョブを投入する。

- 一連のジョブの終了を待って、取りまとめ結果を出力する。

```
top {{
  for $paramA (3, 5, 7, 11, 13) {
    $paramB = 0;
    $ENV{paramA} = $paramA;
    $ENV{paramB} = $paramB;
    do_job {{ paramA=$paramA
              paramB=$paramB }} {{
      system("test_bin $paramA $paramB
              > result.$paramA.$paramB");
    }}
  }
}}
when {{ paramA = $paramA
        paramB = $paramB }} {{
  $result->{$paramA}{$paramB} =
    read_result("result.$paramA.$paramB");
  $paramB =
    optimize($result->{$paramA});
  $ENV{paramA} = $paramA;
  $ENV{paramB} = $paramB;
  if ($paramB) {
    do_job {{ paramA=$paramA
              paramB=$paramB }} {{
      system("test_bin $paramA $paramB
              > result.$paramA.$paramB");
    }}
  }
}}
when {{ paramA=$paramA paramB=* }} {{
    system("cat result.$paramA.*
            > result_A.$paramA");
}}
```

図 4: 動的に追加ジョブ投入を行う例
5 種の paramA のそれぞれに関して最適値を求める。read_result 関数はジョブの出力ファイルを読み、optimize 関数はリスト中の結果を解析して次に投入すべきジョブのパラメータを返す。

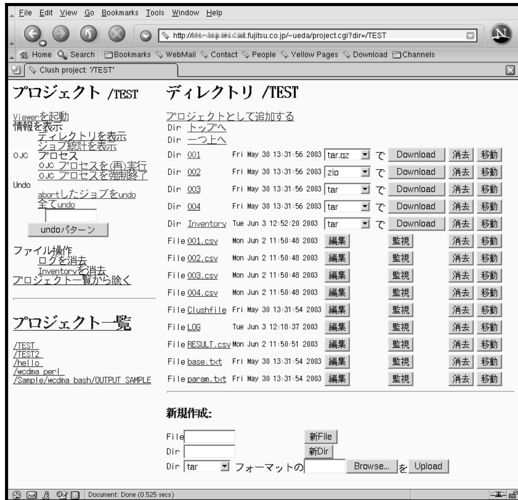

図 5: WWW ポータル

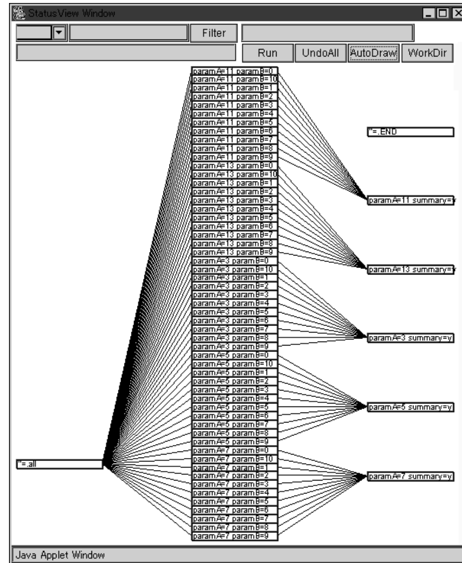

図 6: OJC ビューア

3.3.3 undo および再実行

OJC の名前付きジョブ管理部では、全てのジョブの状態がインベントリに収められており、OJC インタプリタが異常終了したような場合でも、再度 OJC を実行することで、インベントリの情報を用いて終了時点の状態を再構成しそのまま実行を継続することが出来る。

このことを利用すると、インベントリ中のジョブの状態を未投入状態にすることで、そのジョブだけを再投入することが出来る。例えば、`paramA=5` のジョブだけを再投入したい場合は、インベントリからその名前を持つジョブの情報だけを消去すればよい。これを undo と呼ぶ。

undo した後で OJC インタプリタを再度実行すると、最初にインベントリに記録されている完了ジョブの情報を元にローカルスクリプトを全て再実行する。ローカルスクリプトからは、名前付きジョブ投入部も再度呼び出されるが、名前付きジョブ投入部は、消去したジョブだけを再投入する。このため必要なジョブのみを再度実行することができる。undo を用いると、インベントリ情報だけを操作し、OJC スクリプトを一切変更しないため、スクリプトの管理が容易であるという利点がある。

3.4 ポータルとビューア

OJC は、ユーザが WWW ブラウザからジョブ投入などを行えるよう、ポータルサービスを提供している (図 5)。ポータルは、perl で書かれた cgi で実装されている。

ポータル画面からは、ファイルの作成、消去、アップ/ダウンロードなど一般のファイル操作の他、OJC に特化した操作として、OJC の実行、停止、ジョブの undo およびジョブの進行状況と平均ジョブ実行時間の表示などが出来る。また、図 6 のようなビューア画面を起動することが出来る。

ビューアはインベントリ中のジョブの状態および、待ち合わせ関係に基づいて表示を行う。投入されたジョブはビューア中でジョブの待ち合わせ関係を元にツリー状に配置される。

それぞれのジョブは状態に応じて色付けられており、ジョブがキュー中で待っているときは水色、実行中は緑、終了は青などとなっている。そして、ジョブをクリックすることで、ジョブ毎の状態や、標準出力などのファイルを表示することが出来る。

多くのジョブが表示されることが考えられるので、ジョブ名やジョブの状態を元に表示をフィルターする機能も備えている。

4 CAD-Grid への適用と評価

CAD-Grid 環境は、著者らが所属する社内のグリッド環境で、パラメータスウィープを必要とするユーザ向けに構築された環境である [16]。この環境のユーザ向けポータルとして OJC を適用し、評価を行った。

CAD-Grid 環境を利用したのは、移動通信シス

	投入前	実行中	終了後	合計
OJC なし	40 分	10 分	10 分	60 分
OJC あり	12 分	1 分	2 分	15 分

表 1: ジョブを投入する際に必要な平均ユーザ作業時間

テムの設計部門であり、設計パラメータや、伝搬路特性を表すパラメータ合計 27 種類のうちいくつかを変化させながら Bit Error Rate 特性をシミュレーションによって求めた。この実証実験において、ユーザは 4ヵ月の間に 620 個のジョブを OJC を利用して投入した。

同じユーザは以前にも同様の大量ジョブ投入を行っており、その時と OJC を用いた場合のジョブ投入で、ジョブ投入前、実行時およびジョブ完了後にユーザが費した作業時間を計測してもらい、1 ジョブ当たりの平均作業時間を表 1 に示した。

この表から、ユーザの作業時間は約 1/4 に減っているのが分かる。さらにこれらの作業時間の詳細をインタビューした結果、以下のように削減出来た理由がわかった。

- ユーザ作業のうちもっとも大きな効果があったのはジョブ投入前の作業である。従来はジョブを投入する際に、投入すべきパラメータとそのバリエーションに基づいて、パラメータファイルを一つずつ作成し、更に関連したジョブ毎に異なるディレクトリを作成して保存する作業が必要であった。OJC を用いることで、ユーザは初期に投入すべきパラメータと終了条件を与えるだけで動的にジョブ投入できるようになった。
- ジョブ実行中およびジョブ実行後の作業に関しても OJC の効果があった。従来は定期的に (例えば毎朝) ジョブが全て終了したか確認し、終了したものに関しては取りまとめの作業を行っていた。取りまとめの作業にはファイルフォーマットのコンバートも行っていた。OJC スクリプトにより、これらの作業はほぼ全自動で行うことが出来、ジョブ終了をチェックして、終了後にすでに集計の終ったファイルを取り出すのみですむようになった。

また、これらの定量的な評価の他に、定性的な評価として以下のようなコメントを得た。

- 容易にジョブを取り消せるようになったため、気軽にジョブ投入できるようになった。
- 一部のパラメータを変更して、全てのジョブを再実行するのが簡単だったため、飛び込みの再検証依頼に素早く対応することが出来た。
- ケアレスミスが減ったため、最初から細かいメッシュでのジョブ投入が出来た。
- 従来はジョブ投入にかかる作業量がかなり多く、計算機を増やしてもユーザ作業がボトルネックとなって投入量を増やすことが出来なかった。

5 おわりに

グリッド技術により、多くのノードの結合した計算環境が構築できるようになったが、これらのノードを使い切る程多くのジョブを管理する手法は確立していなかった。特に、パラメータスウィープなどの大量ジョブの投入、終了管理とワークフローの記述を統一的に管理する方法がなかった。

著者らは、ジョブを重複しないように投入する名前付きジョブ管理部とジョブの待ち合わせが宣言的に記述できるスクリプト言語の処理系の二つの部分から構成されるオーガニックジョブコントローラを開発した。オーガニックジョブコントローラを用いることにより、大量のジョブ投入、パラメータスィープ、終了待ち、ワークフロー、動的ジョブ生成などが容易に行えるようになった。

シミュレーションジョブを大量に投入する実証実験の結果、ジョブ投入と管理にかかる作業量が 1/4 と大幅に減ることが示され、OJC の有用性が確認された。

謝辞

本研究の一部は新エネルギー・産業技術総合開発機構 基盤技術研究促進事業「高信頼・低消費電力サーバの研究開発」によって行われた。

CAD-Grid 実証実験に関して多くの方の御協力をいただきました。まず、実験の機会と Grid 環境を提供してくださった富士通研究所の門岡氏、今村氏、清水氏に感謝致します。また、実証実験のプラットフォームを用意してくださった富士通生産技術本部の中村氏、山下氏、高橋氏に感謝致しします。最後に、実証実験に協力して評価データをくださった富士通研究所の岩松氏と富士通東日本ディジタルテクノロジの関氏、大川氏、吉田氏に感謝致します。

参考文献

[1] The globus grid project. http://www.globus.org/.

[2] LSF. http://www.platform.com/.

[3] Ninf network server project. http://ninf.apgrid.org/.

[4] OpenPBS. http://www.openpbs.org/.

[5] D. Abramson, R. Sosic, J Giddy, and B. Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing*, Virginia, August 1995.

[6] David Abramson, Jon Giddy, and Lew Kotler. High performance parametric modeling with Nimord/G: Killer application for the global grid? In *International Parallel and Distributed Processing Simposium(IPDPS)*, pp. 520–528, May 2000.

[7] Rajkkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computation grid. In *the 4th International Conference on High Perfomance Computing in Asia-Pacivfic Region (HPC ASIA 2000)*, USA, 2000. IEEE Computer Society Press.

[8] Henri Casanova, MyungHo Kim, James S. Plank, and Jack J. Dongarra. Adaptive scheduling for task farming with grid middleware. *The International Journal of High Performance Computing Applications*, Vol. 13, No. 3, pp. 231–240, Fall 1999.

[9] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *Proceedings of the 9th Heterogeneous Computing workshop (HCW'2000)*, 2000.

[10] High Throughput Computing Team. Condor. http://www.cs.wisc.edu/condor/publications.html.

[11] Adrian DeVivo, Maurice Yarrow, and KIaren M. McCann. A comparison of parameter study creation and job submission tools. Technical Report 002, NASA, 2001.

[12] Dietmar W. Erwin. UNICORE – a grid computing environment. *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 13-15, pp. 1395–1410, 2002. Grid Computing environments Special Issue.

[13] G. C. Fox, D Gannon, and M. Thomas. Editorial: A summary of grid computing environments. *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 13-15, pp. 1035–1044, 2002. http://grids.ucs.indiana.edu/ptliupages/publications/gcesurvey.pdf.

[14] Chris Johnson, Steve Parker, and David Weinstein. Component-based problem solving environments for large-scale scientific computing. *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 13-15, pp. 1337–1350, 2002. Grid Computing environments Special Issue.

[15] 上田晴康, 吉田武俊, 安里彰. ジョブ投入と待ち合わせを記述できるスクリプト言語 オーガニックジョブコントローラ — cad-grid への適用 —. In *PSE Workshop in Sapporo*, 2003 年 7 月.

[16] 山下智規, 中村武雄, 上田晴康, 今村信貴, 岩松隆則, 斉藤直之. グリッド環境「CAD-Grid] 構築と移動通信システムシミュレーションへの適用. In *PSE Workshop in Sapporo*, 2003 年 7 月.

[17] Maurice Yarroa, Karen M. McCann, Rupak Biswas, and Rob F. Van der Wijngaart. An advanced user interface approach for complex parameter study process specification on the information power grid. Technical Report 009, NASA, 2000.