耐故障性を重視したRPCシステム Ninf-Cの設計と実装

中田　秀基 [†,††] 田中　良夫 [†]
松岡　聡 [††,†††] 関口　智嗣 [†]

耐故障性を重視した RPC システム Ninf-C の設計と実装に関して述べる。Ninf-C は、全体として数日から数ヶ月を要する大規模なマスタワーカ型計算を安定して実行することを目的としたシステムで、ウィスコンシン大学で開発されたスケジューリングシステム Condor の提供する機能を利用することで、マスタを含むシステム全体に耐故障性を持たせている。Ninf-C の RPC は、Condor のファイルステージ機能を用いて実現される。直接ソケット通信を使用せずにファイル経由で通信を行うことで、マスタとワーカのチェックポイントをとることを可能とした。また、ファイルに残った通信記録を用いてマスタの状態を復元する。さらに、Condor-G を利用することで、Globus によって構築されたグリッド環境下での運用も可能である。

Ninf-C の有効性を確認するため、クラスタ環境で簡単なマスタワーカ型プログラムを長時間実行した。この際、マスタおよびワーカを実行しているマシンをシャットダウンするといった人為的な外乱をあたえたが、プログラムは 19 時間かけて問題なく実行を終了し、Ninf-C の耐故障性が実証された。

Design and implementation of a Fault-Tolerant RPC system: Ninf-C

Hidemoto Nakada [†,††], Yoshio Tanaka [†,††]
Satoshi Matsuoka [††,†††] and Satoshi Sekiguchi [†]

In this paper, we describe design and implementation of a fault tolerant RPC system, **Ninf-C**. Ninf-C is designed for large-scale master-worker programs, that take from a few days to a few months for its execution. Ninf-C takes Condor, developed by University Wisconsin, as the base structure of the system. It uses file transmission and checkpointing mechanisms and provides system-wide robustness for programmers. In Ninf-C, master and workers communicate each other using file, not the socket, making crash-recovery easy.

To prove robustness of the system, we performed an experiment on a heterogeneous cluster consists of x86 and SPARC. We ran a simple but long-running master-worker program on the cluster and rebooted several machines of the cluster to disturb the program execution. As a result, the program execution finished normally, showing the robustness of Ninf-C.

1. はじめに

グリッドをはじめとする、資源が不均質、不安定な環境下では、効率的に実行できる並列計算の種類が限定される。資源の性能が不均質な環境では、通信相手の計算の遅れによって、自らの計算が遅滞するようなアルゴリズムはうまく機能しない。たとえばバリア同期を繰り返す類の計算アルゴリズムは、1 台低速な計算機が混じるだけで、システム全体の性能が大幅に低下する。また、資源が不安定な環境下では、実行に数週間を要する大規模計算を行う際に、計算に参加するすべての計算機が計算の完了まで動作していることを期待することはできない。この場合は、すべての計算機がクラッシュしたり、リブートされたりする可能性があることを前提にしなければならない。

このような環境下で大規模な並列計算を効率的に行うには、並列計算のアルゴリズムに、1) 計算資源の速度差に寛容、2) 計算の状態が集中していてチェックポイントを取るのが容易、3) 参加計算機の増減に寛容、といった特徴が要請される。

このような性質を持つ計算アルゴリズムの一つにマスタワーカ型計算がある。マスタワーカ型計算とは、マスタがジョブキューを管理し、ワーカが実際のジョブの処理を行う計算である。ワーカは、マスタからジョブを受け取り、計算して、結果をマスタに返す。これをジョブがなくなるまで繰り返す。このモデルは、ワーカを実行する計算機の増減や速度差に対して寛容

† 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)
†† 東京工業大学 Tokyo Institute of Technology
††† 国立情報学研究所 National Institute of Information

である上、計算状態がマスタに集中するため、チェックポイントが比較的容易に実現できる。多くの応用問題がマスタワーカ型計算で実行できることが知られている。

われわれは、RPC(Remote Procedure Call) がプログラミングモデルとして、マスタワーカに適していると考え、RPC を実装したシステムを提案してきたが、[1),2)] これらのシステムは、高速な実行を指向した結果、耐故障性が限定されていた。

本稿ではより耐故障性を重視した RPC システム Ninf-C の設計と実装に関して述べる。Ninf-C は、全体として数日から数ヶ月を要する大規模なマスタワーカ型計算を安定して実行することを目的としたシステムで、ウィスコンシン大学で開発されたスケジューリングシステム Condor[3),4)] をベースとする。Condor の提供する機能を利用することで、マスタを含むシステム全体が耐故障性を持つ。

Ninf-C はプログラマに RPC(Remote Procedure Call) を実現する API と、呼び出される関数のインターフェイスを記述するための IDL を提供する。プログラマは非同期の RPC を用いることで、容易に複雑な構造を持つマスタワーカ計算を記述することができる。

本稿の構成は次のとおりである。2 節で Condor の概要を説明する。3 節で本稿で提案する Ninf-C システムの設計と実装について述べる。4 節で、実行例を示して評価を行う。5 節で、関連研究との比較を行う。6 節で結論と今後の課題を述べる。

2. Condor の概要

Condor は、米国ウィスコンシン大学のグループが開発したハイスループットコンピューティングを指向したキューイングシステムである。当初はキャンパス内の遊休計算機を有効利用することを目的としていたが、現在では、Globus Toolkit[5)] を用いて構築したグリッド環境のメタスケジューラとしても広く使用されている。

Condor は、計算可能な計算機の集合を Condor プールと呼び、ユーザがサブミットしたジョブに対して、Condor プールに属する計算機に割り当て、実行する。このジョブに対して計算機を割り当てる方法に、マッチメイキングと呼ばれる方法が使用される。さらに、計算が適宜チェックポイントされ、サーバフェイル時の再実行や優先順位の高いジョブによるプリエンプションがサポートされていることが、Condor の特徴である。

2.1 Condor の構造

図 1 に condor システムの概要を示す。Condor に参加するノードには、セントラルマネージャ、サブミットマシン、実行マシン の 3 つの役割がある。1 つのノードが複数の役割を兼ねることも可能である。セントラルマネージャは、システム全体で 1 つでなければならないが、サブミットマシンの数は制限されていない。

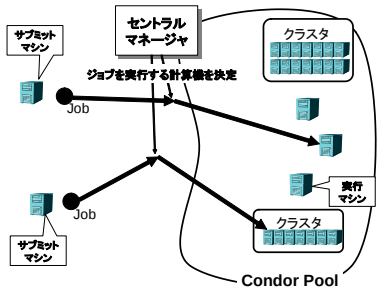

図 1 Condor の概要

ユーザは Condor のシステムコールライブラリとリンクされたジョブを用意する。次にジョブを記述したサブミットファイルを用意し、コマンドを用いてサブミットマシンから投入する。サブミットマシンはサブミットされたジョブの情報を、セントラルマネージャに定期的に送信する。実行マシンでは、Startd と呼ばれるデーモンが自らの状態をモニタし、セントラルマネージャに送信する。セントラルマネージャはこれらのジョブと実行マシンの情報を比較して、適切なマッチングを選び出し、実行マシンにジョブを割り当てる。

2.2 チェックポイントとリモートシステムコール

Condor は、システムコールをフックすることで、チェックポイントとリモートシステムコールを実現するライブラリを提供している。このライブラリをリンクしたバイナリを Condor で実行すると、バイナリの行うファイル入出力がフックされ、サブミットマシン上で実行される Shadow プロセスによって代行される。したがって、バイナリはサブミットマシンのファイルシステムにアクセスすることになる。

さらに、定められた間隔でチェックポイントが行われる。チェックポイントファイルは指定されたチェックポイントサーバ、もしくはサブミットマシンに置かれる。さらに、チェックポイントを利用したプリエンプションやマイグレーションもサポートされている。

3. Ninf-C の設計と実装

3.1 耐故障性への要請と設計の指針

長時間に及ぶマスタワーカプログラムを安定して効率よく実行するには、以下が要請される。

(1) マスタがチェックポイントされ、マスタ実行計算機がクラッシュした際には、自動的にチェックポイントからリスタートされること

(2) ワーカもチェックポイントされ、ワーカを実行している計算機がクラッシュ、もしくは計算機

プールから取り除かれた場合には、他の計算機でリスタートすることができること

(3) ワーカを実行可能な計算機が増えたらそれを適宜利用できること。

これらの機能は、第 1 項を除いてほとんどすべて Condor によって提供されている。しかし次項で述べるようにソケットを用いた通常の RPC 実装を行うと、利用できる機能が制約されてしまう。したがって、Ninf-C の設計においては Condor の提供する機能を最大限利用できるように工夫した上で、上記第 1 項の機能を追加することとした。

3.2 ファイル経由の通信

前項でも述べたとおり、長期間に及ぶマスタワーカ計算を安定して実行するには、マスタとワーカのそれぞれをチェックポイントできなければならない。ここで問題になるのが、通信である。もっとも標準的な通信手段はソケットによる方法だが、ソケット通信を行うプログラムに対してチェックポイントを行うことは一般に困難であり、多くのチェックポイントライブラリでは対応していない。Condor のチェックポイントライブラリも例外ではない。

そこで Ninf-C では、マスタとワーカの通信をファイルを介して行うことにした。通信内容は一度ファイルに書き出されてから、別のプログラム (Condor) によって転送される。このようにすることで、マスタとワーカのプロセスを安全にチェックポイントすることができる。もちろん実際に転送を行うプロセスはチェックポイントできないが、転送プロセスは内部に状態をもたないので問題はない。

また、ファイル経由で通信を行うことの副作用として、すべての通信の記録がマスタ側にファイルとして残ることになる。この通信記録を利用することで、マスタプログラムの内部状態の復元が容易になる。これに関しては 3.6 で詳しく述べる。

3.3 システムの概要

Ninf-C の RPC は、Condor のファイルステージ機能を用いて実現されている。マスタプログラムはサブミットマシン上で動作する。ワーカとなる計算関数はファイルでの通信を実現するライブラリとリンクして、リモート実行ファイルとして用意しておく。このリモート実行ファイルを、マスタから Condor を通じてサブミットすることで RPC を行う。図 2 にこの様子を示す。

マスタプログラムから RPC が実行されると、Ninf-C のランタイムは、送信すべき引数をマーシャリングしてひとつのデータファイルに書き出す。次に、このデータファイルをステージファイルとして転送するように指定したサブミットファイルを生成する。このサブミットファイルでは、実行ファイルとして呼び出された RPC に対応するリモート実行ファイルを指定する。さらに、RPC の返り値を記述する出力ファイルをマスタ側にステージアウトするように指定する。そして、Condor のジョブサブミットコマンドである `condor_submit` を実行して、このジョブをサブミットする。

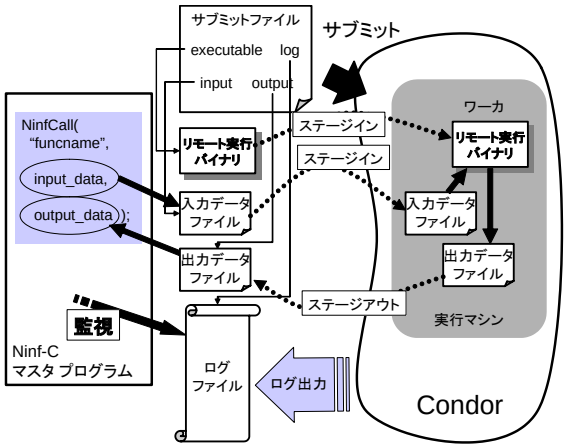

図 2 Ninf-C の概要

次に Condor のログファイルを監視し、ジョブのサブミットを知らせるログが出力されるのを待ち、そのログから Condor でのジョブ ID を取得する。

Condor にサブミットされたワーカは、Condor プール内のいずれかの実行マシン上で実行されることになる。この際に、リモート実行ファイルとともに、入力データファイルも実行マシン上に転送される。実行マシンは、リモート実行ファイルを起動する。リモート実行ファイルは、入力データを読み込んでアンマーシャリングして C のデータ構造に変換し、それらを引数として、計算ルーチンを起動する。計算ルーチンの実行が終了したら、結果のデータをマーシャリングして出力データファイルに書き出す。Condor システムは、リモート実行ファイルの実行が終了すると、出力データファイルをサブミットマシンに転送し、ログファイルにそのジョブが終了したことを書き出す。

マスタは当該ジョブが終了したことを知らせるログが出力されるまで、sleep を繰り返しながらログファイルの監視を続けている。探しているログが出力されたら、ジョブが終了し、出力ファイルが書き出されているはずなので、出力ファイルをオープンしてその内容を読み出してアンマーシャリングし、RPC 呼び出しの際に引数として与えられていた返り値用の配列に収める。

3.4 Ninf-C の API

Ninf-C の提供する API のセマンティクスは、Ninf[1] の提供する API とほぼ同じであるが、名前付けのコンベンションと引数の一部が若干異なる。表 1 に Ninf-C の API 関数の一部を示す。

もっとも重要な関数は NinfCall と NinfCallAsync

```
Module pi;
Define pi_trial(
IN  int seed, IN  int n,
OUT double *ratio)
"tries n random points"
Required "pi_trial.o"
{
  extern double pi_trial(int seed, int n);
  * ratio = pi_trial(seed, n);
}
```

図 3 IDL ファイルの例

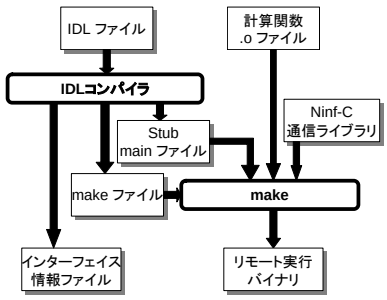

図 4 IDL コンパイラ

である。これらの関数は、第一引数で指定されたリモート関数を呼び出す。前者は関数呼び出しが終了するまでブロックするのに対して、後者はブロックせずにリターンする。その際に第二引数で指定された変数に、各呼び出しを識別するセッション ID を納める。

各呼び出しの終了を待つには、その呼び出しに対応するセッション ID やセッション ID の集合を指定して、NinfWait 系の関数を呼び出す。さまざまな用途を考慮し、5 通りの NinfWait 系関数が用意されている。

3.5 Ninf IDL と IDL コンパイラ

Ninf-C を利用するには、リモートで実行したい関数のインターフェイスを定義する必要がある。これには Ninf IDL(Interface Description Language) と呼ぶ言語を用いる。Ninf-C の使用する Ninf IDL の仕様は、Ninf[1]、および Ninf-G[2] とほぼ同じであるためここでは割愛し、簡単な例を図 3 に示すにとどめる。

Ninf-C は IDL をコンパイルして、インターフェイス情報ファイル、および計算関数をラップして実行ファイルを作成するための、スタブ main ファイルを生成する。さらに、スタブ main ファイルと計算関数と Ninf-C の通信ライブラリをリンクして実行ファイルを作成することを支援する Make ファイルを出力する。この様子を図 4 に示す。

3.6 マスタのチェックポインティング

Condor は、リモート環境で実行するプログラムに関してはチェックポイント機能を提供している。したがってワーカは適宜チェックポイントが行われ、必要であればマイグレーションも行われる。

しかし、システム全体としての耐故障性を得るためにはワーカ側だけでは不十分で、マスタ側に関しても

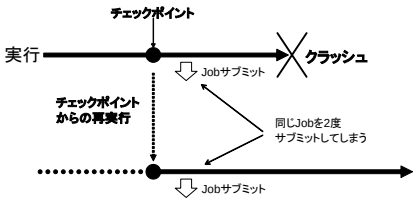

図 5 マスタのチェックポイント

対策する必要である。Condor はサブミットマシン上でジョブを実行する Scheduler ユニバースを提供しているが、このユニバースではリカバリー時のジョブの再起動機能は提供されるものの、チェックポイントは行われない。

また、マスタの場合は定期的にチェックポイントを取るだけでは不十分である。直前のチェックポイントを行った後、ジョブのサブミットを行い、その後にサブミットマシンがクラッシュした場合を考えてみる。単純にチェックポイントした時点から再スタートするだけでは、すでにサブミットしたジョブを再度サブミットすることになってしまう。これは計算の結果には影響を与えないが無駄である (図 5)。

Ninf-C では、これら問題を解決するために、チェックポイントライブラリを用いた定期的なチェックポイントと、2 重サブミット防止機構を実装している。

まず、マスタプログラムを Condor のチェックポイントライブラリとリンクし、チェックポイントを可能にする。これを単純に Scheduler ユニバースでサブミットするだけでは定期的にチェックポイントはされないし、クラッシュ後にはプログラムの先頭から再実行されてしまう。このため、マスタプログラムを簡単なスクリプトプログラムでラップし、このスクリプトを Scheduler ユニバースに投入する。スクリプトは、マスタプログラムに定期的にシグナルを送り、チェックポイントファイルを作成させる。クラッシュ後の再起動時には、チェックポイントファイルの存在を確認し、存在する場合にはチェックポイントファイルからの再起動を行う。

この動作をサポートするために、Ninf-C はジョブ投入用のスクリプト、`ncrun` を提供する。`ncrun` は、マスタプログラムを Condor の scheduler ユニバースに投入するためのサブミットファイルと、マスタプログラムをラップするスクリプトを自動的に生成し、生成したサブミットファイルを用いて、生成したラッパスクリプトをサブミットする。

また、マスタからの 2 重サブミットを防止するために、サブミットファイルを作成する前に、サブミットファイルの存在チェックを行っている。各 RPC 呼び出しに対応するサブミットファイルは、マスタプログラム全体でシリアルな番号に応じた名称がつけられる。新たにジョブをサブミットしようとした場合には、シリアル番号からサブミットファイル名を決定し、サブ

表 1 Ninf-C API

型	意味
NinfErrorCode	エラーコード。実態は int
NinfSessionId	セッションを識別するための ID。実態は int
関数	**意味**
`int NinfParseArg(int argc, char ** argv);`	引数配列を用いて初期化を行う。
`NinfErrorCode NinfFinalize();`	Ninf-C ランタイムを終了する
`NinfErrorCode NinfCall(char * entry, ...);`	entry で指定されたリモート関数の同期呼び出しを行う。
`NinfErrorCode NinfCallAsync ( char * entry, NinfSessionId * pSessionId, ...);`	entry で指定されたリモート関数の非同期呼び出しを行い、そのセッション ID を *pSessionId に収める
`NinfErrorCode NinfWait(NinfSessionId id);`	引数で指定されたセッションの終了を待つ。
`NinfErrorCode NinfWaitAll();`	これまでに実行されたすべてのセッションの終了を待つ。
`NinfErrorCode NinfWaitAny(NinfSessionId * id);`	これまでに実行されたすべてのセッションのうちいずれかのセッションの終了を待つ。

```
Executable   = exec.$$(OpSys).$$(Arch)
Requirements = \
((Arch=="INTEL" && OpSys=="LINUX") || \
 (Arch=="SUN4c" && OpSys=="SOLARIS29"))
```

図 6 複数のアーキテクチャを自動的に選択するサブミットファイル

ミットファイルを作成する。この際に、すでに同名のサブミットファイルがないか確認し、あればすでに (クラッシュする前に) サブミットしていると判断して、サブミットを行わない。

3.7 ヘテロアーキテクチャの利用

Ninf-C では、マスタプログラムとリモート実行ファイル間の通信に用いるファイルはすべて XDR でエンコードされている。このため、リモート実行ファイルを実行するアーキテクチャが、マスタのアーキテクチャと同じである必要はない。

Condor は複数のアーキテクチャのサーバを自動的に選択し、適切なバイナリをサーバに提供する方法をサポートしている。図 6 に、intel アーキテクチャの LINUX もしくは、SPARC の SOLARIS 2.9 を選択してジョブをサブミットする場合のサブミットファイルの一部を示す。この場合、実行バイナリは、`exec.LINUX.INTEL` および `exec.SOLARIS29.SUN4c` という名前で用意しておかなければならない。

Ninf-C は、この機構を利用して複数のアーキテクチャのプロセッサを同時に利用した実行を行う。Ninf-C の IDL コンパイラの出力する make ファイルは、make 実行時に、Condor の識別するアーキテクチャ、OS 情報を `condor_status` コマンドで自動的に取得し、バイナリ名に付加するように構成されている。バイナリ名は `_stub_(モジュール名)_(エントリ名).(OS名).(アーキテクチャ名)` となる。

Ninf-C のマスタモジュールは実行時にカレントディレクトリを検索し、`_stub_(モジュール名)_(エントリ名)` で始まる実行属性のついたファイルを収集する。このファイル名から OS 名とアーキテクチャ名の情報を取得する。この情報をサブミットファイルの Requirements フィールドに反映することで、複数アーキテクチャの同時利用を実現している。

3.8 スロットリング

Ninf-C プログラムから RPC 呼び出しを多数回繰り返し行うと、その回数だけの Condor サブミットが行われる。Condor はサブミット時に実行ファイルをコピーしてキューイングする。このファイルは比較的大きいため、極端に多数回のサブミットを行うとサブミットマシンの資源が圧迫される。

Ninf-C では、これを避けるために、コンフィギュレーションファイルで同時にサブミットするジョブの数に制約を加えている。これを超えるジョブをサブミットしようとすると、システムがブロックし、実行中のジョブの終了を待つ。実行中のジョブが所定の数以下になると、実行を再開する。

3.9 Ninf-C での実行の手順

Ninf-C でマスタワーカプログラムを実行する手順は以下のようになる。

(1) Condor を適切にセットアップする。
(2) Ninf-C の提供するコンパイルドライバ `nccc` を用いて、マスタプログラムをコンパイル、リンクする。
(3) ワーカ用のリモート実行ファイルとインターフェイス情報ファイルを作成する。まず、ライブラリのインターフェイスを IDL で記述し、それを IDL コンパイラ `ncgen` でコンパイルし、生成されたた Makefile で make を行う。
(4) 実行用コンフィギュレーションファイルを用意する。このファイルには、インターフェイス情報ファイルのパスやスロットリングのための最大同時サブミットジョブ数を記述する。
(5) 実行用のスクリプト `ncrun` を用いてマスタプログラムを実行する。

4. 実行例と評価

Ninf-C のアーキテクチャヘテロ環境への対応と、対故障性を確認するために実験を行った。さらに、大規

```
double * ratios = new double[n];
int    * ids   = new int[n];
for (int i = 0; i < n; i++){
  int id;
  NinfCallAsync("pi/pi_trial", &(ids[i],
                i, m, &(ratios[i]));
}
NinfWaitAll();

double ratioSum = 0.0;
for (int i = 0; i < n; i++);
  ratioSum += ratios[i];
printf("pi = %f\n", (ratioSum / n) * 4.0);
```

図 7 マスタプログラムの一部

模環境での有効性を確認するための実験を行った。

4.1 ヘテロ環境への対応と耐故障性の評価

アーキテクチャヘテロ環境への対応と、対故障性を確認するために、アーキテクチャヘテロな Condor プールを構成し、簡単なマスタワーカ型プログラムを長時間実行した。この際に、サブミットマシンおよび実行マシンのシャットダウンなど、人為的な外乱をあたえ、対故障性を確認した。

4.1.1 実 験 環 境

実験に用いた Condor プールは 5 台の x86 の LINUX と 7 台の SPARC Solaris の 2 種類の計算機で構成されている。LINUX ノードは Pentium III 1.4GHz を 2PE ずつ搭載している。Solaris ノードの PE は 450MH z である。7 台中の、2 台は 1PE、2 台は 2PE、3 台は 4PE である。セントラルマネージャは、LINUX ノードのひとつで稼動している。マスタを実行するサブミットマシンとしては、2PE の Solaris ノードのひとつを用いた。なお、これらの計算機はすべて外部からの Globus Gatekeeper を介した使用に対して公開されており、実験中もさまざまなジョブが投入されている。

4.1.2 使用プログラム

評価には、モンテカルロ法を用いて円周率を求めるプログラムを用いた。このプログラムは、正方形の中に乱数で点を生成し、その点が正方形に内接する円に含まれるかどうかを調べることで円の面積を算出、面積から円周率を逆算するものである。

図 7 にマスタプログラムの一部を示す。`NinfCallAsync` で非同期に n 回の RPC 呼び出しを行い、`NinfWaitAll` ですべての終了を待っている。その後、`ratios` に蓄えられた各呼び出しにおける割合から円周率を導出する。マスタから呼び出される関数の IDL を図 3 に示す。

1 つのワーカで、2×10^{10} 個の点を発生させ判定を行った。さらにこのワーカの実行を 200 回行い、総計 4×10^{12} 回の試行を行った。ワーカの 1 度の実行時間は Pentium III 1.4GHz のマシンでは 50 分程度。SPARC 450MHz のマシンでは 3 時間程度である。

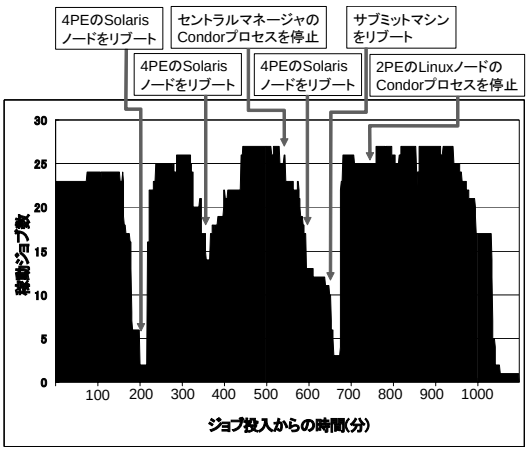

図 8 稼動ジョブ数の推移

4.1.3 実験の概要

長時間実行による下記の操作を外乱として与えた。

- 約 210 分後 4PE の Solaris ノードをリブート
- 約 350 分後 4PE の Solaris ノードをリブート
- 約 540 分後 セントラルマネージャの Condor ジョブを停止
- 約 600 分後 4PE の Solaris ノードをリブート
- 約 660 分後 サブミットマシンをリブート
- 約 740 分後 LINUX ノードの 1 つで Condor ジョブを停止

4.1.4 実験の結果

図 8 に、実行中の稼動ジョブ数の推移を示す。横軸は開始後の時間を分で示している。縦軸は稼動ジョブ数である。このデータは、Condor の出力するログファイルから取得した。

外乱操作を与えるたびに、稼動ジョブ数が大きく低下するものの、自動的に回復していることがわかる。特に、サブミットマシンをリブートした際にはマスタプロセスも一度停止、破棄されているが、本稿で示した手法によって自動的に状態を回復して、処理を続行していることに注意されたい。

なお、800 分から 900 分のあたりで、外乱操作を行っていないときにもジョブ数の細かい変動が見られるが、これらは Condor 以外の手法によるジョブサブミッションによって、Condor のジョブが退避しているものであると思われる。

4.2 大規模環境での有効性の評価

Ninf-C の大規模環境での有効性を確認するために、大規模な Condor プールを用いて長時間のジョブを実行した。

4.2.1 実 験 環 境

実験環境としては東工大の Titech Grid を用いた。Titech Grid は東工大キャンパスの各所に設置した比較的小規模のクラスタを専用の高速ネットワークで接続した計算環境である。構成計算機としては、一部の

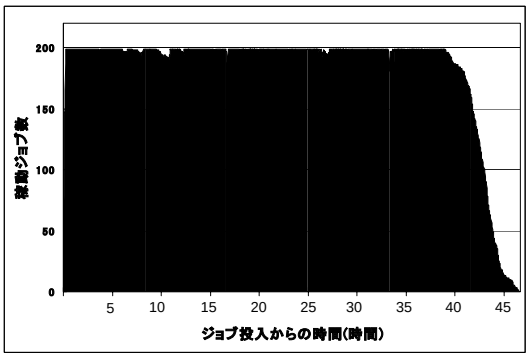

図 9 分子動力学シミュレーションの稼動ジョブ数の推移

ノードを除き 2CPU の Pentium III 1.4 GHz のブレードサーバを用いており、総 CPU 数は 800 台に及ぶ。実験時、Condor プールにはこのうちの 734 プロセッサが登録されていた。他のユーザに対する過度の侵食を防止するため、最大同時実行プロセッサ数を 200 に制限してプログラムを実行した。

4.2.2 実験の概要

アプリケーションプログラムとしては、分子動力学シミュレーションプログラムである Amber6 の Sander を用いた。オリジナルの Sander はファイルから設定情報を読み込んで動作するが、これを一部改変して、Ninf-C を用いてマスタ側からさまざまな設定情報をコントロール可能にした。

このプログラムを用いて、初期速度の異なる分子動力学シミュレーションを、1000 回行った。初期速度は Sander 内部でランダムに生成されるが、その乱数シードをマスタ側から制御している。対象分子は、20 残基からなる小タンパク質である Trp-Cage を用い、シュミレーション時間は 360 マイクロ秒とした。相互作用パラメータとしては parm99[6] を用いた。

1 つのシミュレーションにかかる時間は、他の負荷がまったくない場合には、Pentium III 1.4 GHz のノードで 7 時間程度である。計算機を占有していないため、負荷状態に応じてこの時間は大きく変動し、もっとも長いものでは 12 時間程度かかっていた。

4.2.3 実験の結果

図 9 に、実行中の稼動ジョブ数の推移を示す。ほぼ全域で上限の 200 プロセッサを使用できていることがわかる。また、全実行時間には 2 日間弱がかかっている。これにより、Ninf-C が大規模環境での長時間の実行でも安定して動作することが確認できた。

5. 関連研究

5.1 Condor DAGMan

DAGMan(Directed Acyclic Graph manager)[7] は、依存関係を持つ複数のジョブのフローを Condor 上で実行する機構である。ジョブのフローは DAG ファイルと呼ばれるファイルに記述する。各ジョブは Condor のサブミットファイルで表され、DAG ファイルには、個々のサブミットファイル間の依存関係が記述される。

DAGMan 本体は Ninf-C のマスタと同様に、scheduler ユニバースのジョブとしてサブミットマシンで実行される。DAGMan は、DAG ファイルを参照して次に実行するジョブを決定し、そのジョブのサブミットファイルを Condor に発行する。ジョブの進行状態は Condor のログファイルを監視して把握する。

DAGMan はジョブの実行状況以外の状態を持たない。ジョブの実行状況はログファイルから再構築できるため、チェックポインティングを行わずに耐故障性を実現できる。

5.1.1 Ninf-C との比較

DAGMan と Ninf-C の構造は類似しているが、いくつか相違点がある。DAGMan では Condor のサブミットファイルをユーザが記述しなければならないため、ユーザに Condor に関する深い知識が求められるのに対し、Ninf-C では、基本的には Condor に関する知識を必要としない。

また、DAGMan では、静的に記述されたワークフローの固定的な実行しかできない。たとえばループ構造のワークフローを構成して、収束するまで計算を繰り返す、といったことは DAGMan では難しい。これに対して、Ninf-C では通常のプログラムの一部として Condor 上の計算を組み込むことができるため、プログラマが容易に、任意のワークフローを構成して、実行することができる。

5.2 Condor MW

MW(Master Worker)[8] は、Condor を利用した汎用のマスタワーカアプリケーション向けフレームワークで C++で記述されている。ユーザはシステムの提供する、ドライバ、ワーカ、タスクの 3 つのクラスを継承したクラスを作成する。ドライバがマスタプログラムに相当し、Scheduler ユニバースでサブミットマシン上で実行される。

MW では、通信手法として次の 3 つがサポートされている。PVM、TCP/IP ソケット、ファイルである。最も多用されているのは耐故障性を提供するファイルによる通信である。ファイル通信は、Condor のリモート IO 機能を用いて実現される。

5.2.1 Ninf-C との比較

MW と Ninf-C は、双方ともマスタ・ワーカ型のアプリケーションを対象としているが、ワーカとジョブのマッピング方法が異なる。Ninf-C は、ワーカの 1 度の呼び出しを Condor の 1 つのジョブとして実現しているのに対して、MW は 1 つのジョブで多数回のワーカ呼び出しを処理する。MW の手法では、ワーカ呼び出しの際のオーバヘッドが少ないが、ワーカが実際には呼び出されていない間も、プロセッサをジョ

ブが占有してしまう。これに対して Ninf-C の手法では、ワーカが呼び出されていない間はプロセッサを開放する。

このデザインチョイスの相違は、それぞれのシステムが想定する、プログラムの構造とワーカの粒度に起因する。Ninf-C は、単純なパラメータスイープ型のアプリケーションだけでなく、比較的複雑な同期構造を含むマスタ・ワーカを対象とし、ワーカの 1 度の実行時間を数時間と想定している。同期構造を含むプログラムでは、新たなワーカを呼び出すことのできない、空き時間が生じる可能性があり、その際の空き時間の長さはワーカの 1 回の実行時間に比例する。この長大な空き時間の間、ジョブがプロセッサを占有することがないよう、Ninf-C ではジョブをワーカの 1 度の呼び出しに対応させている。

5.3 Ninf-G

Ninf-G[2] は、Globus toolkit[5] をベースとして実装された RPC システムである。API としては、Global Grid Forum で標準化された GridRPC API[9] を用いている。

Ninf-G は比較的細粒度のマスタ・ワーカ型アプリケーションを、大規模なグリッド環境で実行することに主眼をおいている。このため、一度のワーカの実行、すなわち RPC 呼び出しに掛かるオーバヘッドを縮小することに力点をおいて、API およびシステム構造が設計されている。その反面、耐故障性はそれほど重視されておらず、上位レイヤでの再実行による耐故障性は考慮されているもののマスタ側の故障にはまったく対応できない。

6. おわりに

本稿では、長時間に及ぶマスタワーカプログラムの実行を目的とし対故障性を重視した RPC システム Ninf-C の設計と実装に関して述べた。Ninf-C は Condor の提供する機能を利用して、ファイル経由の RPC を行い、マスタを含むシステム全体に対して対故障性を提供する。Ninf-C を用いたシステムの耐故障性を確認するため、クラスタ環境で、人為的な外乱を与えつつ長時間の実行を行った。その結果、高い耐故障性が確認できた。また、200 プロセッサを同時に用いた実験を行い大規模環境での有効性を確認した。

今後の課題としては以下が挙げられる。

- 実アプリケーションでの有効性の検証
  今回評価に用いたプログラムは、単純なマスタワーカ型プログラムであった。今後は、より複雑な構造を持つ実アプリケーションを用いて、評価を行う必要がある。具体的には、レプリカ交換法や遺伝アルゴリズムによる最適化を考えている。
- マスタ側チェックポイント機能の改良
  3.6 で提案した、2 重サブミットを防止する機能はシステム全体が決定的に動作することを前提としており、非決定的に動作するアプリケーションには対応できない。これに対応するには、すでにサブミットされているジョブをキャンセルする機構と、クライアントプログラムの内部からチェックポイントのタイミングを制御してキャンセルしなければならない事態を最小限に抑える機構が必要になる。

参 考 文 献

1) Nakada, H., Sato, M. and Sekiguchi, S.: Design and Implementations of Ninf: towards a Global Computing Infrastructure, *Future Generation Computing Systems, Metacomputing Issue*, Vol. 15, No. 5-6, pp. 649–658 (1999).
2) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *Journal of Grid Computing*, Vol. 1, No. 1, pp. 41–51 (2003).
3) Livny, M., Basney, J., Raman, R. and Tannenbaum, T.: Mechanisms for High Throughput Computing, *SPEEDUP Journal*, Vol. 11, No. 1 (1997).
4) Condor. http://www.cs.wisc.edu/condor/.
5) Foster, I. and Kesselman, C.: Globus: A metacomputing infrastructure toolkit., *Proc. of Workshop on Environments and Tools, SIAM.* (1996).
6) Wang, J., Cieplak, P. and Kollman, P. A.: How well does a restrained electrostatic potential (RESP) model perform in calculating conformational energies of organic and biological molecules?, *Journal of Computational Chemistry*, Vol. 21, No. 12, pp. 1049–1074 (1999).
7) DAGMan. http://www.cs.wisc.edu/condor/dagman/.
8) Goux, J.-P., Kulkarni, S., Linderoth, J. and Yorde, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid, *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsuburgh, Pennsylvania, pp. 43 – 50 (2000).
9) 中田秀基, 田中良夫, 松岡聡, 関口智嗣: Grid RPC システムの API の提案, 情報処理学会研究報告 HPC, Vol. 2001, No. 78, pp. 37–42 (2001).