耐障害性を考慮したNinf-Gアプリケーションの実装と評価

谷村勇輔[†] 池上努[†] 中田秀基[†,††]
田中良夫[†] 関口智嗣[†]

我々はグリッドのアプリケーションにとって耐障害性が重要課題であることを踏まえて，タスク並列アプリケーションを GridRPC システムの 1 つである Ninf-G を用いて実装し，アジア太平洋地域のグリッドのテストベッド上で長時間にわたり実行した．その中で障害パターンを集めて，障害の検知や復旧にかかるコストを測定しながら耐障害性の機構を検討した．本研究により，グリッドが持つ不安定さに対応したアプリケーションやミドルウェアでは，障害検知や復旧の操作におけるタスク実行の性能低下に留意する必要があるとともに，性能低下を防ぐために，障害検知のためのタイムアウト値の最小化や復旧のバックグラウンド処理，障害を考慮したタスク割り当てが必要であることが分かった．こうして，グリッドのアプリケーション開発者に対して開発や実行時の留意点を示すとともに，GridRPC の枠組の上位に求められる耐障害に関する機構の設計への指針を示した．

Implementation of Fault-Tolerant Ninf-G Application

Yusuke Tanimura,[†] Tsutomu Ikegami,[†] Hidemoto Nakada,[†,††]
Yoshio Tanaka[†] and Satoshi Sekiguchi[†]

Recently, fault-tolerance is one of the big issues for the Grid application. In this paper, the task parallel application is implemented with Ninf-G which is a GridRPC system and experimented on the Asia-Pacific Grid testbed for a long term. Typical fault patterns were gathered and costs of fault detection and recovery were measured in order to discuss a better fault-tolereant mechanism. This study reveals a polar lesson for application and middleware to avoid decline of task throughput by fault-tolerant controlled operations, such as timeout minimization for fault detection, background recovery and duplicate task assignments. This paper also shows a steer for design of the automated fault-tolerant mechanism in a higher layer of the GridRPC framework.

1. はじめに

近年，大規模な科学技術計算を行うためにグリッド環境が注目されている．ネットワーク越しに計算資源を使うことによるオーバーヘッドは小さくないが，同じ計算を複数の異なるパラメータで実行するパラメータ探索型や，1 つの計算を複数のタスクに分割して行うマスタースレーブ型の計算を行うアプリケーションは比較的スケーラビリティが高く，グリッド環境を有効に利用できると期待されている．また，Globus Toolkit[1] に代表されるミドルウェアの発展により，システムアーキテクチャや設定の違いは吸収されつつあり，実際的なアプリケーションの実行段階に入っている．

一方，アプリケーション・ユーザにとって，ネットワークの突然の停止や定期／不定期に行われるシステムのメンテナンスは，長時間のアプリケーションの実行を阻害するものである．利用するグリッド環境の安定性の向上や不安定さを吸収できるミドルウェアの開発は重要な課題の 1 つであるが，実際にどういう障害が発生するか，どういう対応が可能であるかの検討はこれまで十分になされてきたとはいえない．そこで本研究では，タスク並列で計算を行うアプリケーションに耐障害性の機構を組み込み，アジア太平洋地域のテストベッドを利用して長期間の実験を行う．そして，実際に起きる障害を分析し，耐障害性の機構を改良しながら，障害検知のコストや障害からの復旧手順について検討する．こうして得られた知見をまとめ，アプリケーション開発者に対して実装に際して留意するべき点を提供するとともに，ミドルウェアのレベルで障害を吸収するための指針を得ることを目指す．

本研究では，アプリケーションの例題として，量子化学計算の 1 つである TDDFT（Time-Dependent Density

† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology
†† 東京工業大学
Tokyo Institute of Technology

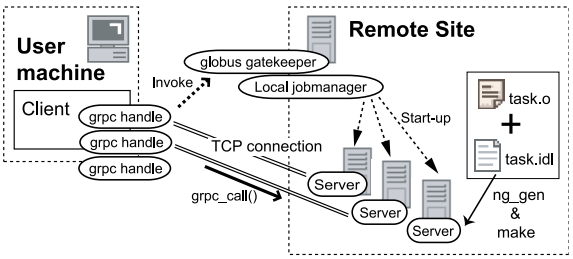

図 1 Ninf-G のクライアント・サーバ間接続

Functional Theory)[2] を用いる．TDDFT は，ホットスポットとなる計算部を数百規模の並列タスクに分割して処理し，それを数千回繰り返すことでユーザが必要とする精度の結果が得られるアプリケーションであり，問題規模に応じた長時間の実行が必要である．

アプリケーションの実装には，Globus Toolkit の上に構築されたミドルウェアである Ninf-G[3] を用いる．本論文で用いた Ninf-G のバージョンは，2004 年 9 月 10 日に公開された 2.2.0 である．Ninf-G は，RPC の仕組みをグリッドに拡張した GridRPC[4] のプログラミングモデルを提供する．GridRPC はタスク並列のアプリケーションを容易に実装できるよう，GGF（Global Grid Forum）[5] にて，その API の標準化が検討されている．Ninf-G では，これまでレプリカ交換モンテカルロ[6] や気象予測シミュレーション[7] がアプリケーションとして実装され，基本性能やスケーラビリティに関するシステム評価がなされているが，耐障害性については十分な評価がなされていない．また，Ninf-G では障害対応をアプリケーションレベルで記述することになっており，本実験を通して，それをミドルウェアで対応するための課題を明らかにする．

2. Ninf-G と耐障害性

グリッド環境ではアプリケーションを長期間実行する過程において，予期する／しないに関らずネットワークが切断されたり，計算機の停止が起きることは避けがたい．こうした障害に対して，アプリケーションの実行をやり直すことなく継続できることが望まれ，グリッドのミドルウェアがそれを支援することが期待されている．アプリケーション・ユーザの手を全く煩わせないためには，ミドルウェアレベルの完全な自動復旧が考えられる．それに対して，Ninf-G Version 2 がサポートする GridRPC API ではプリミティブな API がまず提案され，障害に関しては上位の API やミドルウェアでサポートすることが考えられている．プリミティブな API では適切に障害や異常を検知して，それを呼び出したプログラムに通知するよう設計されている．本論文ではこれを用いて，アプリケーションレベルで障害に対応できる機構を実装する．

2.1 GridRPC API のエラーハンドリング

Ninf-G のアプリケーションは，そのプログラムの中で GridRPC の API を呼ぶことにより，プログラマが意図した計算を遠隔の計算資源で実行する．GridRPC はクライアント・サーバ型に基づいており，遠隔の計算資源にてサーバプロセスを実行する．以降，本論文ではサーバプロセスをサーバ，また対応する Ninf-G のクライアントプロセスをクライアントと記す．

Ninf-G のクライアント・プログラムの流れとして，最初にサーバと 1 対 1 で対応するハンドルを作成し，その後，そのハンドルを介して呼び出したい関数をハンドルに対応するサーバで実行する．非同期に関数を呼び出した場合には，呼び出し時に取得したセッション ID を使って実行の完了を待つことができる．最後に全てのハンドルを解放し，サーバを終了する．この一連の流れにおいて，クライアントはサーバとの TCP 接続を維持する（**図 1**）．つまり，正常にアプリケーションが動作するためには，これらの TCP 接続が正常に開始され，維持されることが必要となる．逆にいえば，ネットワークが停止しても計算機が停止しても，この TCP 接続は異常となるため，サーバの障害は TCP 接続を監視することによって検知される．

表 1 に Ninf-G が提供する主な GridRPC API と，これらを呼び出してエラーに遭遇した際に，戻り値であるエラーコードから検出できる障害の内容を示す．エラーコードは，GGF でもまだ検討段階であるが，Ninf-G ではドラフトをもとにそれを実装している．表に示したように，異なる障害が同一のエラーコードに対応し，エラーコードから障害の内容や原因を分析することはできない．エラーコードは状況を示しており，アプリケーション開発者は状況に対してどう対応するかを記述する．例えば，GRPC_COMMUNICATION_FAILED は RPC の実行中にクライアントとサーバの TCP 接続が切断されたことを意味する．以降，クライアントは，接続を失ったサーバのハンドルを再作成しない限り，そのサーバに対して RPC を実行できない．そこで，開発者は状況に応じてサーバを再起動したり，あるいは別の計算機でサーバを起動したりするようクライアントのプログラムを作成することになる．

2.2 ハートビート機能

障害に対応したエラーハンドリングに加えて，Ninf-G はハートビート機能を提供する．これはサーバからクライアントに対して定期的にパケットを送信する機能であり，これより，問題なくサーバが動作しているかや TCP 接続が維持されているかをクライアント側で確認できる．つまり，サーバの暴走やサーバからの一方的なネットワークの切断が生じた時に，クライアントでの検知が困難な場合が考えられるが，そういう場合でもクライアントがハングアップ

表 1　Ninf-G の主要な API において検出される障害と対応するエラーコード

API	Possible fault detection	Error code
grpc_function_handle_init()	DNS query failure	GRPC_SERVER_NOT_FOUND
	Server machine is down.	GRPC_SERVER_NOT_FOUND
	Network to server is down.	GRPC_SERVER_NOT_FOUND
	Globus-gatekeeper is not running.	GRPC_SERVER_NOT_FOUND
	GRAM invoking fails at authentication.	GRPC_OTHER_ERROR_CODE
grpc_function_handle_destruct()	–	–
grpc_call()	TCP disconnection during data transfer	GRPC_COMMUNICATION_FAILED
	RPC failure to disconnected server	GRPC_OTHER_ERROR_CODE
grpc_call_async()	TCP disconnection during data transfer	GRPC_COMMUNICATION_FAILED
	RPC failure to disconnected server	GRPC_OTHER_ERROR_CODE
grpc_cancel()	–	–
grpc_wait_any()	TCP disconnection (for nonblocking data transfer)	GRPC_SESSION_FAILED
	Hearbeat becomes timeout.	GRPC_SESSION_FAILED

してしまうのを防ぐという 2 次的な手段として利用できる．ハートビート機能は，別の条件において，ファイアーウォール越しの TCP 接続を維持するためにも利用される．

ハートビートの利用において，アプリケーション実行者はハートビートのパケット，あるいは計算結果が一定時間届かないことに対するタイムアウト時間を設定する．タイムアウトしたサーバとの接続は切断され，そのサーバに対応するハンドルは無効となる．API では，タイムアウトが発生すると grpc_wait_any() に代表される待機関数にて GRPC_SESSION_FAILED のエラーコードが返される．これは，データ転送に失敗した際のエラーコードと同様である．

3. 障害を考慮した実装例

本研究において，我々は TDDFT（Time-Dependent Density Functional Theory）[2] 方程式をグリッドのアプリケーションとして Ninf-G を用いて実装し，かつ耐障害性の機構を組み込んだ．TDDFT は，分子レベルのシミュレーション手法の 1 つであり，多電子励起状態をあらわに取り扱うための量子力学に基づく手法である．TDDFT では，ホットスポットとなる計算部を単純なタスク並列計算として実装できる．**図 2** に簡易化した計算の流れを示す．これは 1）入力データの読み込みとパラメータの設定，2）Ninf-G を使ったタスク並列の計算，3）クライアントでの逐次計算に分けられ，最初に 1）を行った後，2）を実行して 3）を実行する繰り返しを指定された回数だけ行う．

Ninf-G では，Globus Toolkit の MDS（Monitoring and Discovery System）を利用して呼び出したい関数が実行可能な計算機を検索し，そこでサーバを起動する．しかし，現在提供されている GridRPC API では，タスクを割り当てたサーバをアプリケーションが管理することになる．障害を考慮した実装では，これに加えて障害が発生しているサーバも管理しなければならない．また，障害が発生したサーバをすみやかに復旧する機構を実装する必要がある．

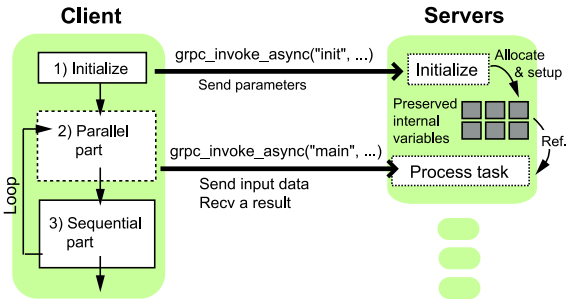

図 2　Ninf-G を使った TDDFT の実装

以降の節では，基本的なサーバの状態管理とタスク割り当てを説明し，その上で障害を想定した状態管理と復旧手順について述べる．

3.1 サーバの状態管理とタスク割り当て

図 2 に示すように，2）の各タスクを実行するためには，実行するサーバにおいて，分子データが記述された入力ファイルを読み込み，与えられたパラメータを使って配列の初期化を行う 1）の処理が必要となる．本論文では，1）の初期化を行う関数を「初期化メソッド」，2）のタスクを実行する関数を「メインメソッド」と呼ぶ．初期化メソッドで生成された配列は，Ninf-G のリモートオブジェクトの機能を使って状態が保存され，以降で呼ばれるメインメソッドで参照可能である．リモートオブジェクトは，同一サーバにおける複数のセッションをまたいでデータをサーバに保存することができる Ninf-G の Version 2 で追加された機能である．特に，今回の TDDFT の並列モデルでは，全てのサーバが同じように初期化されるため，1 度，初期化メソッドを完了したサーバではメインメソッドを繰り返し実行することができる．つまり，この TDDFT の単純な実行手順は次のようになる．最初に，利用する全サーバで初期化のメソッドを実行する．その後，メインメソッドを非同期に呼び出し，タスクを完了したサーバに対して

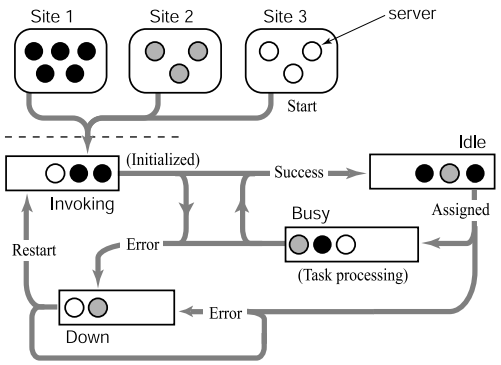

図 3 Ninf-G サーバの状態遷移

次のタスクを与える．ただし，時間ステップを更新する毎に，そのステップにおける全タスクが完了するのを待つ必要がある．

これを踏まえて，本プログラムでは，ハンドル作成関数にて起動した Ninf-G のサーバを**図 3** と以下に示すような状態遷移に基づいて管理する．

a) 各サイトで起動されたサーバは Invoking の状態に置かれる．
b) 初期化メソッドが完了すると Invoking の状態から Idle の状態に移る．
c) Idle の状態のサーバに対してタスクが割り当てられ，タスクの実行中は Busy の状態となる．
d) タスクの実行を完了すると Idle の状態に移る．

3.2 障害発生時の対応と復旧手順

次に，障害発生時の対応と障害からの復旧について述べる．障害が発生するとクライアントとサーバ間の TCP 接続が切断され，それは 2.1 節で述べた通り，GridRPC API のエラーコードから知ることができる．そこで，API が障害を示すエラーコードで返ると，図 3 に示したようにサーバを Down の状態に遷移させる．Down の状態にあるサーバに対しては，タスク実行が依頼されないように実装する．

一方，Ninf-G では，サーバは TCP 接続の切断とともに自動的に終了している．それをアプリケーションのレベルで復旧するためには，そのサーバのハンドルを破棄し，再作成することでサーバを再起動するようプログラムしなければならない．さらに今回は，リモートオブジェクトを利用しているため，ハンドルを再作成した後に初期化メソッドの実行が必要である．これらが全て正常に終了するとサーバは Idle の状態に移行し，復旧が完了したことになる．本研究で実装する TDDFT では，全てのサーバに対する初期化メソッドは同一であり，不変である．そこで，あらかじめ初期化メソッドを登録しておき，Down の状態にあるサーバを復旧させる時に自動的に呼び出すこととした．

復旧の操作は，1 時間おきに各サーバの状態を一通り確認し，全てのサーバが Down の状態にあるクラスタに対して行う．これは次の理由による．Ninf-G では，サーバの起動において Globus Toolkit の GRAM（Grid Resource Allocation Manager）を利用し，実際のプロセスは PBS や LSF などのローカルなスケジューラから起動される．GRAM が行う GSI（Grid Security Infrastructure）の認証やスケジューラを介したプロセス起動はオーバーヘッドが少なくないため，通常，1 度の GRAM 呼び出しで複数のサーバを起動する方法がとられる．しかし，この方法では一部のサーバが Down の状態にあっても，それに対応する CPU は解放されない．バッチジョブが残っているためである．つまり，Ninf-G クライアントが特定のサーバだけを再起動するのは困難であり，正常なサーバも含めた再起動が必要となるのである．もちろん，クラスタ内の全サーバが Down の状態になるまで待たずに，生存サーバ数がある一定値以下になった段階で再起動を行う方法も考えられる．ただし，一定値は後述する復旧のコストを考慮して決めるべきと考える．

4. テストベッドを利用した耐障害性の実験

耐障害性を検討するために，前章で述べたように実装した TDDFT のプログラムを ApGrid[8]/PRAGMA[9] のテストベッドを用いて断続的に実行した．実験は 2004 年 6 月 1 日から 8 月 31 日までの 3ヶ月にわたり行い，その期間中に AIST☆1，SDSC☆2，KISTI☆3，KU☆4，NCHC☆5，USM☆6，TITECH☆7，NCSA☆8，UNAM☆9，BII☆10の 10 サイトの資源を利用し，その中で，どのような障害が発生し，Ninf-G の障害検知の機構がそれらに対して正常に動作したか，復旧に要するコストなどを調査した．そうして得られた結果をもとに望ましい耐障害性の仕組みについて検討を行った．

4.1 長時間の実行結果

実験期間中，本プログラムの合計実行時間は 906 時間（約 38 日）であった．同時に Ninf-G サーバを起動した最大サイト数は 7 つ，最大サーバ数（CPU 数）は 67 であった．最長実行時間は，2004 年 7 月 28 日 20 時より 8 月 4 日 16 時までの 164 時間（約 7 日）であり，以降ではその結

☆1 産業技術総合研究所
☆2 San Diego Supercomputer Center（USA）
☆3 Korea Institute of Science and Technology Information
☆4 Kasetsart University（Thailand）
☆5 National Center for High-performance Computing（Taiwan）
☆6 Universiti Sains Malaysia
☆7 東京工業大学
☆8 National Center for Supercomputing Applications（USA）
☆9 Universidad Nacional Autónoma México
☆10 Bioinformatics Institute（Singapore）

表 2 最長実行時間を記録した際の Ninf-G サーバの計算環境

Site	#CPU	System	Throughput
AIST	28	PentiumIII 1.4GHz	116 MB/s
SDSC[1]	12	Xeon 2.4GHz	0.044 MB/s
KISTI[2]	16	Pentium4 1.7GHz	0.28 MB/s
KU[3]	2	Athlon 1GHz	0.050 MB/s
NCHC[4]	1	Athlon 1.67GHz	0.23 MB/s

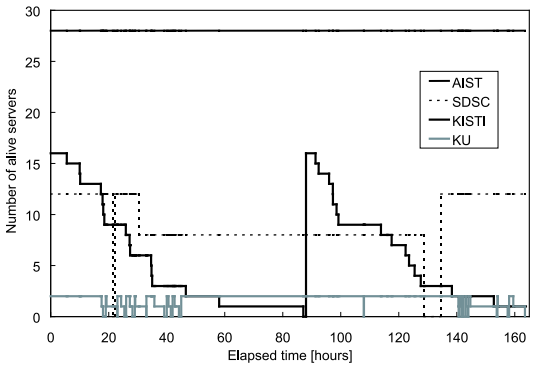

図 4 サイト別生存サーバ数の変化

果の詳細を述べる．最長実行時間を記録した際の Ninf-G サーバを実行した計算環境は**表 2** に示す通り，5 サイト 59 サーバであった．Ninf-G クライアントは AIST の計算機で動作させ，サイト毎にローカルのスケジューラを使って一斉に起動した．表中のスループットは，実験前に AIST の計算機から 1MB のメッセージを TCP で送信することで計測した．一方，TDDFT のシミュレーション対象とした分子は，ループ毎に 122 のタスクに分割され，1 タスク当たりクライアントからサーバへ 4.87MB のデータを送信し，3.25MB の結果を受信するものであった．

図 4 に NCHC を除いたサイト別の生存サーバ数の履歴を示す．クライアントからサーバが Down の状態にあると認知された理由は以下の通りであった．1）ネットワークのスループットが低くなった際にデータ転送に失敗した．2）サーバまでのネットワークが完全に切断した．3）電力不足のために計算機が停止された．4）TCP 接続がハングアップした．切断の ACK を受信できなかったものと思われる．1）〜3）の障害は TCP 接続の切断により検知され，4）の障害はサーバからのハートビートがタイムアウトすることで検知された．SDSC のサーバが停止した原因は主に 2）であったため，同時に全てのサーバが停止した．それに対して，KISTI や KU のサーバは主に 1）や 4）の原因により停止したため，段階的にサーバが減少している．また，KU の停止頻度は他のサイトに比べて高いといえる．

次に，1 時間おきの処理タスク数を**図 5** に示す．130 時間付近でタスクの実行が途切れているのは，サーバの復旧

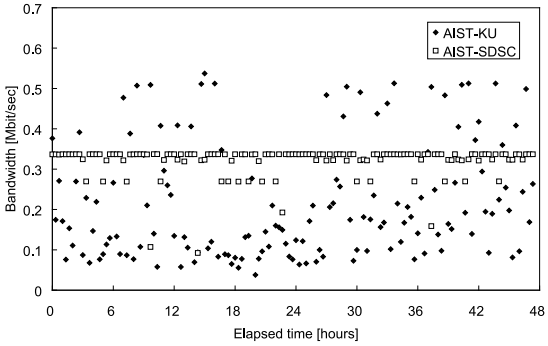

図 6 ネットワーク性能の変化

を試みたクラスタが混雑しており，サーバが起動できるまで待っていたためであり，改善が必要である．一方，本実験は並列化による速度向上を議論することを目的としていなかったため，1 タスクあたりの実行時間が小さく，LAN で接続された AIST の資源を使って処理されるタスクが多くなる．そのため，障害が発生してリモートの資源が使えない間のほうが，処理タスク数が大きくなる傾向がある．しかし，生存サーバ数が頻繁に変化している時間帯において処理タスク数が減少しており，問題である．この原因は 2 つ考えられる．1 つは，作成したプログラムではループ毎に全タスクの終了を待つため，終了待ちのタスク数が少なくなった段階での障害はボトルネックになる可能性が高い．ハートビートのタイムアウトによって検知される障害であれば，そこで実行していたタスクが失敗となるまで，最大でタイムアウト値と同じ 300〜400 秒の時間がかかる．もう 1 つは，障害サイトの復旧はクライアントがタスクの実行を停止して，逐次的に行うためである．この問題の対策は 4.3，4.4 および 4.5 節にて論じる．

4.2 テストベッドのネットワークの安定性

図 4 では，表 2 に示したスループットがそれほど大差がないにも関わらず，SDSC と KU の計算機で障害の頻度が異なる．そこで，TCP 接続の切断の状況と原因をさらに調査した．**図 6** に 20 分毎に計測した AIST-SDSC 間，AIST-KU 間のネットワークのスループットを示す．図より，明らかに AIST-KU 間のスループットは時間の経過とともに大きく変動し，時折，その性能は非常に悪くなることが分かる．

さらに，AIST でクライアントを実行し，AIST と SDSC から 4 つずつをサーバを選んで 20 時間実行した場合と，AIST と KU から同様にサーバを選んで実行した場合におけるクライアントからの TCP の再送頻度を計測した．前者は 0.014%であったのに対し，後者は 0.11%と 8 倍近く異なる．これらの結果より，長時間の実験で遭遇した TCP 接続の切断が，ネットワークの安定性が低いために生じた

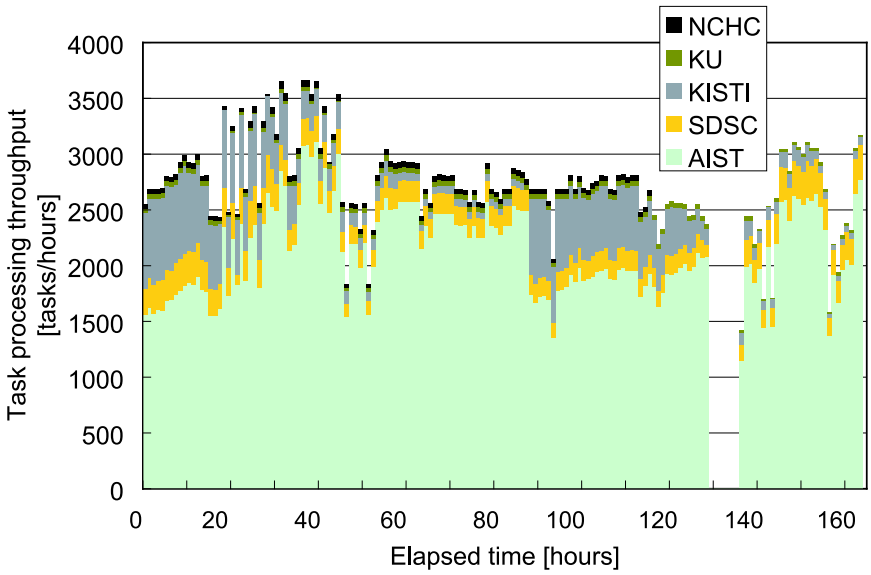

図 5 時間毎のサイト別処理タスク数

と考えることができる．

4.3 障害検知のためのコスト

データ転送中のエラーにより TCP 接続が切断されるなど正常にソケットが閉じられた場合，Ninf-G は速やかに障害を検知できる．しかし，先に述べたように，ハートビートがタイムアウトすることで障害を検知する場合は，タイムアウトまでの時間を待つことになる．一方，タイムアウト値を短く設定すると小さな遅延も異常と判断され，接続が切られてしまう．そこで，適切なタイムアウト値を設定するために，ハートビートがどの程度遅れて到着しうるかを実験した．

表 3 は AIST と SDSC，KU の各 4 サーバを用いて本プログラムを実行し，ハートビートを受信する間隔を計測している．計測では AIST と SDSC のサーバは 60 秒毎にハートビートを送信し，KU のサーバは 80 秒毎に送信する．その送信間隔が 5 回過ぎる間にサーバから何もデータを受信しない場合，すなわち 300 秒あるいは 400 秒間データを受信しない場合，接続はタイムアウトエラーとなる．ただし，ハートビートの送信は計算データの転送と同じ TCP 接続を利用するため，データ転送中にハートビートを受信できない．かつ，計算データが到着するとハートビートが届いたことと同等の扱いとなる．そこで，表 3 に入力データの転送に要する時間の平均値と最大値も示す．

表より，SDSC や KU のサーバからのハートビートは遅延しているが，ハートビートの送信間隔（60 秒または 80 秒）とデータ転送時間の最大値を加えた値より，最も長いハートビートの受信間隔は短いことが分かる．つまり，ハートビートのタイムアウト値はデータ転送時間より少し大きく設定し，ハートビートの送信間隔を短く，送信回数を増やすように設定することで，タイムアウトを待つコストを抑えることができる．

表 3 ハートビートの受信間隔と入力データの転送時間 [sec]

	Heartbeat interval		Data transfer time	
Site	Ave.	Max.	Ave.	Max.
AIST	30.3	60.0	0.396	0.476
SDSC	30.6	81.0	19.4	25.7
KU	65.1	203	96.7	159

4.4 復旧のためのコスト

4.4.1 復旧コストの測定

図 5 の結果より，復旧のためのコストが単位時間当たりのタスクの処理能力に影響を与えることが明らかとなったため，復旧コストを正確に見積もり，最小限のコストになるよう復旧のタイミングや頻度を考える必要がある．そこで，AIST，SDSC，KU の 3 つのサイトにおける復旧コストを測定した．復旧手順と各段階で要する時間を**表 4** および以下にまとめた．

復旧を行うには，復旧するサーバに対応するハンドルを一度解放しなければならない．ただし，先にも述べたように，クラスタ毎にサーバを一斉起動している場合，同時に起動した正常なサーバの再起動も必要となる．このハンドルの解放と正常なサーバの停止が表 4 の 1）である．次に停止したサーバを再起動する要求を出す．これが 2）に相当し，計算資源へアクセスでき，かつ認証を終えてローカルのスケジューラに要求を出せたかを確認する．3）はサーバが正常に起動し，クライアントに通知する操作である．

表 4 復旧に要するコスト [sec]

	AIST	SDSC	KU
1) Server halt	0.00709	0.00465	0.854
2) Server check	0.556	2.37	1.67
3) Server restart	4.83	10.6	2.80
4) Initilization	6.74	3.67	48.1
Total time	12.1	16.6	53.5

表 5 バックグラウンドで復旧処理を行った場合の性能 [sec]

No fault	215.62
Background recovery	216.26
Foreground recovery	263.35

4）は再起動されたサーバが初期化メソッドを完了する操作である．このコストはアプリケーションや計算機性能に依存する．1）〜4）を足し合わせたのが合計時間である．

表 4 に示す結果は 3 回試行の最良値である．KU はネットワークの状況によって，示した時間よりもずっと大きなコストがかかる可能性がある．表 4 では 3）と 4）の操作に大きな時間がかかっている．しかし，これらはサーバ側の処理であるため，クライアントはオーバーラップして他の処理を行い，復旧コストを隠蔽することが望ましい．復旧を試みても，依然として計算資源へのアクセスができないなど復旧に失敗する場合は 2）でエラーとなる．つまり，復旧を頻繁に試みると 2）のコストが積み重なることになるが，これもスレッドなどを利用することによりバックグラウンドで処理可能である．

4.4.2 復旧コストの隠蔽

前項の検討をもとに，1）〜4）の復旧手順をまとめて 1 つのスレッドとし，バックグラウンド処理した際の性能を評価した．外乱の多いネットワークや計算機の性能差による影響を取り除くため，実験では AIST のクラスタにおいて 6 サーバを 2 組起動した．1 ループ目のタスク並列の計算途中に 1 組を停止させ，そのループが終了してから復旧操作を行った．フォアグラウンドで処理する場合は 4）も各サーバに対して逐次的に行い，全ての復旧操作が完了してから次の逐次計算部分に実行が移る．バックグラウンド処理を行う場合は，次の逐次計算がオーバーラップして行われ，その間にサーバの復旧が完了する．計算の開始から 2 回目のループが終了するまでの時間を比較した結果を**表 5** に示す．バックグラウンド処理を行った時の実行時間は，障害が起きない時の実行時間とほぼ同じであり，その有効性が示せたといえる．

4.5 考　　察

ApGrid/PRAGMA のテストベッドを用いた実験では，ネットワークの処理能力が低下したことによる障害が多かった．長期間にわたる実験の結果，それらの障害に対応すべくアプリケーションの開発・実行において注意すべき項目，改善すべき項目を以下に挙げる．

- ネットワークが不安定な環境では，障害検知のためにハートビート機能は有効である．ただし，データ転送に要する時間に基づいて適切なタイムアウト値を設定する必要があり，自動化の余地がある．
- 初期化メソッドを含んだ障害からの復旧コストは非常に大きくなる可能性がある．これを隠蔽するために，障害からの復旧はバックグラウンドで行うべきである．
- アプリケーションは，障害が一時的なものか長期的なものかを知る手段がないため，定期的に復旧を試行することになる．1 回の試行につき，ある一定のコストが必要になるが，多くの場合，全体に占める割合は小さく，またバックグラウンドでの処理も考えられる．
- クラスタにおいてサーバを一斉に起動した場合，障害が発生した一部のサーバをどのタイミングで再起動するかは課題である．正常なサーバを再起動する間，それが利用できなくなる時間と復旧可能かもしれないサーバを利用していない時間とのトレードオフが存在する．スケジューラが提供する機能と合わせて検討する必要がある．
- 処理すべきタスク数がサーバ数に比べて少ない場合，障害対策のために複数のタスクを投入することは有効である．特に，本研究で作成したプログラムは，障害が起きたサーバのタスクを復旧せず，タスクを別のサーバに投入して再計算させる仕組みであるため，その効果が期待できる．

5. 関 連 研 究

RPC と同等の機能をグリッドで実現し，かつ耐障害性を備えたミドルウェアとしては Condor MW[10] や Condor をベースとした Ninf-C[11] が挙げられる．これらのシステムはミドルウェアのレベルでチェックポイント機能を実現し，障害が発生しても実行中のタスクを途中から再実行できるよう設計され，計算機間でタスクをマイグレーションすることも可能である．また，GridRPC API をサポートした別の実装である NetSolve[12] も途中からのタスクの再実行は行わないものの，失敗したタスクを自動的に NetSolve の別のサーバに割り当て直す仕組みを提供する．これらに対して，Ninf-G はタスク割り当てに情報サービスを用いた仕組みを提供していないため，本研究で実装したようなアプリケーションレベルでのサーバの状態管理や障害対応の仕組みが必要であった．しかし，GridRPC の上位に求められる耐障害性に関する機能や API を考える上で，プリミティブな API のエラーハンドリングを用いて耐障害性の機構を検討することは必要である．

また，今回の実験で検出したいくつかの障害は，クライアントとサーバ間の接続を維持する Ninf-G の実装に依存するものである．Ninf-G が接続を維持する理由は，比較的粒度の小さなタスク並列処理もターゲットに置いているためである．本論文では，GridRPC の実装技術として TCP 接続の維持に関する問題は議論の対象としないが，NetSolve のように接続を維持しない実装になっている場合には，タスクの実行中にネットワークが一時遮断されてもプログラムの実行に影響を与えないこともある．しかし，その場合でもデータ転送中のネットワーク障害への対応やハートビートを用いたサーバの状態確認機構は必要である．長時間にわたって効率的に資源を確保し，障害検知やクライアントから行うサーバの復旧に関するコストがタスク実行のスループットに与える影響を抑えるために，本研究で得られた知見は，そうしたシステムにとっても有益な情報となる．

6. 結　　論

本論文では，タスク並列アプリケーションとして実装できる TDDFT を例題として用いて，長時間実行を必要とし，そのために耐障害性を必要としているグリッドのアプリケーションやミドルウェアの開発や実行における留意点を明らかにした．

特に，アプリケーションの開発においては，GridRPC API のエラー処理と Ninf-G サーバの状態管理を適切に行うプログラムを作成し，Ninf-G のリモートオブジェクトを利用した際の障害からの復旧手順を示した．ApGrid/PRAGMA のテストベッドを利用した実験を通して，障害が発生した場合でも計算資源を効率良く維持し，タスク実行のスループットを最小限に抑えるためにスムーズな障害検知と復旧が必要であること，そのためには障害検知のためのタイムアウトを最小化し，復旧処理をバックグラウンドで実行し，障害を考慮したタスク割り当てを行う必要があることを示した．これらを通して，アプリケーションの開発者に耐障害性の機構の実装や実行時の留意点を示し，GridRPC の枠組の上位に求められる耐障害の機構の設計への指針を明らかにした．

今後は，今回得られた知見をもとにミドルウェアにおける耐障害の機構や API を検討していく予定である．

謝辞　本研究を行うにあたり，TDDFT のプログラムを提供して下さった信定（分子研），矢花（筑波大）両氏，貴重なご意見を頂いた武宮（産総研）氏に深く感謝いたします．

本研究は ApGrid ／ PRAGMA における研究活動の一環として行われた．これらの参加研究機関，特に実験に計算資源を提供頂いた機関に感謝いたします．

なお，本研究の一部は文部科学省「経済活性化のための重点技術開発プロジェクト」の一環として実施している「超高速コンピュータ網形成プロジェクト（NAREGI: National Research Grid Initiative）」によるものである．

参 考 文 献

1) Foster, I. and Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit, *Supercomputing Applications and High Performance Computing*, Vol. 11, No. 2, pp. 115–128 (1997).
2) Yabana, K. and Bertsch, G. F.: Time-Dependent Local-Density Approximation in Real Time: Application to Conjugated Molecules, *Quantum Chemistry*, Vol. 75, pp. 55–66 (1999).
3) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *Grid Computing*, Vol. 1, No. 1, pp. 41–51 (2003).
4) Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing, *Proceedings of 3rd International Workshop on Grid Computing* (Parashar, M.(ed.)), pp. 274–278 (2002).
5) GGF: http://www.gridforum.org/.
6) 池上努, 武宮博, 長嶋雲兵, 田中良夫, 関口智嗣: Grid：広域分散並列処理環境での高精度分子シミュレーション–C_{20} 分子のレプリカ交換モンテカルロ, 情報処理学会論文誌 コンピューティングシステム, Vol. 44, No. SIG11, pp. 14–22 (2003).
7) 武宮博, 田中良夫, 中田秀基, 関口智嗣: Ninf-G2：大規模 Grid 環境での利用に即した高機能，高性能 GridRPC システムの実装と評価, 情報処理学会論文誌 コンピューティングシステム (2004).
8) ApGrid: http://www.apgrid.org/.
9) PRAGMA: http://www.pragma-grid.net/.
10) Goux, J., Kulkarni, S., Linderoth, J. and Yoder, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid, *Proceedings of HPDC-9*, pp. 43–50 (2000).
11) 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, 先進的計算基盤システムシンポジウム（SACSIS）論文集 (2004).
12) Casanova, H. and Dongarra, J.: Netsolve: A Network Server for Solve Computational Science Problems, *Supercomputing Applications and High Performance Computing*, Vol. 11, No. 3, pp. 212–223 (1997).