Vol.2009-HPC-121 No.22
2009/8/5

グリッドRPCシステムのクラウド環境への適用

中田秀基[†1]　田中良夫[†1]　関口智嗣[†1]

Amazon EC2 に代表される HaaS 型のクラウドサービスは，ハードウェアそのものをネットワーク越しに従量課金で提供するもので，主にベンチャー企業の IT サービス基盤として広く普及しつつある．しかし，科学技術計算の分野では，現在のところあまり普及していない．この理由の一つとして，アプリケーションやミドルウェアのデプロイと設定が容易でないことが挙げられる．本稿では，クラウド側での設定をまったく行わずに，マスターワーカ型の科学技術計算を実現する機構を提案する．これまでに開発した GridRPC システム Ninf-G5 のジョブ起動機構を改良し，クラウド上のリソース制御機構を組み込んだ．この機能を用いることで，既存のマスタワーカ型の並列プログラムをまったく変更せずにクラウドを用いて大規模並列実行を行うことができる．クラウド側でのデプロイ，設定作業はまったく必要ない．このシステムを用いて簡単な評価実験を行った結果，計算量の大きいマスタワーカ型計算の HaaS 型クラウド上での実行にメリットがあることを確認した．

Application of a Grid RPC System on a Cloud Environment

Hidemoto Nakada ,[†1] Yoshio Tanaka [†1]
and Satoshi Sekiguchi [†1]

HaaS-stlye Cloud services, such as Amazon EC2, which provide virtualized hardware resources charging run-as-you-go base, are now becoming popular as IT service infrastructure mainly for venture business area. However, in the scientific computing area, Cloud services are not well accepted yet. One of the reason is that HaaS is not easy to use, as it requires application and middleware deployment. To cope with the problem, we propose zero-configuration master-worker computing environment on the cloud for scientific computing, which requires no deployment / configuration on the cloud side. We developed a job invocation plug-in capable of controling Amazon EC2 cloud for GridRPC system Ninf-G5. End-users do not have to modify their programs at all. We evaluated the system and confirmed that Cloud can be beneficial for some class of master-worker applications.

1. はじめに

Amazon EC2 (Elastic Computing Cloud)[1] に代表される，HaaS(Hardware as a Service) 型クラウドサービスは，仮想計算機技術を利用し，ネットワーク越しにハードウェア (実際には仮想計算機) 資源をユーザに提供する．HaaS は，比較的低レベルの機能を提供するため，一般のユーザにとっては敷居が高いが，反面どのような運用も可能な柔軟性を持つことから，主にベンチャー企業が Web アプリケーションを提供するための，IT サービス基盤として広く普及しつつある．

一方，科学技術計算の分野では，クラウドは普及しているとは言いがたい．これには幾つかの理由があると考えられる．以下に列挙する．

(1) 個々の仮想計算機が比較的低速．
(2) 複数ノード間のネットワーク速度が低速なため，密に相互通信するアプリケーションに適さない．
(3) 多くのクラウドサービスはアジアに拠点をもたず，国内拠点との通信レイテンシが大きく，スループットも低い．
(4) ミドルウェア，アプリケーションの設定が煩雑で，一般ユーザには敷居が高い．

これらのうち，(1),(2) はメッセージパッシングで記述される密結合を前提とした科学技術計算アプリケーションに対しては大きな障害となる．したがって，このようなアプリケーションのクラウド上での実行は現実的ではない．しかし，マスタワーカ型で記述できる比較的粗結合な並列アプリケーションにとっては，(1),(2),(3) は大きな問題にはならない．従って，残る (4) を解決すれば，クラウド上での活用が普及する可能性がある．本稿では，上記 (4) を解決し，粗結合マスタワーカ型計算を，エンドユーザがクラウド上で容易に行うことができるシステムを提案する．提案システムは，GridRPC システム Ninf-G5 に対するアドオンモジュールであり，クラウド側に対しては，ソフトウェアのデプロイや設定などを行う必要がない．また，クライアントプログラムは，クラスタを用いる場合とまったく同じプログラムを，変更すること無く利用することができる．

次節以降の構成は次の通りである．2 節で研究の背景について述べる．3 節で GridRPC システム Ninf-G について述べる．4 節で提案システムの設計と実装について述べる．5 節で評価を行う．6 節で関連研究に言及し，7 節にまとめを示す．

†1 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)

2. クラウドと Amazon EC2

2.1 クラウドの類別

一般にクラウドと呼ばれるサービスは，下記の 3 つのタイプに大別できる．

- SaaS (Software as a Service) 型: アプリケーションサービスそのものをユーザに提供する．クラウドサービスプロバイダとアプリケーションサービスプロバイダは分離されない．例:salesforce.com, Gmail
- PaaS (Platform as a Service) 型: アプリケーション構築フレームワークをサービスとして提供する．サービス利用者は，アプリケーションサービスプロバイダであり，エンドユーザではない．例: Google App Engine
- HaaS (Hardware as a Service) 型: ハードウェアそのものをサービスとして提供する．サービス利用者は，アプリケーションサービスプロバイダであり，エンドユーザではない．例: Amazon EC2

2.2 Amazon EC2

ここでは，HaaS 型のクラウドサービスの一つで，本稿が対象とする Amazon EC2 を概説する．Amazon EC2 を対象として選んだのは，現在最も普及していること，後述するようにインターフェイスがデファクトスタンダードとなりつつあることによる．

Amazon EC2 (Elastic Computing Cloud)[1] は書籍のインターネット販売で有名な米国 Amazon 社が提供する，代表的な HaaS 型クラウドサービスである．EC2 は，仮想計算機技術を用いて，ユーザに対して仮想的なハードウェアを提供する．ユーザは，特定の OS イメージを指定して，仮想計算機 (インスタンスと呼ぶ) を起動する．起動したインスタンスのルートアカウントには，あらかじめ登録しておいた ssh の公開鍵が設定される．ユーザは，ssh を用いてルートとしてログインし，インスタンスを自由に利用することができる．

EC2 の特徴は，以下の通りである．

- 利用価格が時間に対する従量性で，最も小規模構成で 1 時間 0.1 ドルと安価．不使用時にはまったくコストがかからない．計算ユニットの性能は，2007 年の Xeon に換算して 1.0 から 1.2GHz 程度である．通信にもコストがかかり 1 ギガあたり上り 0.1 ドル，下り 0.17 ドル (使用量に依存) である．
- web サービスインターフェイスで制御可能．さまざまな言語に API が提供されている．コマンドラインからも制御可能．公式の Firefox のプラグインからも管理が可能．なお，このインターフェイスと互換のインターフェイスを持つ，オープンソース実装 Eucalyptus[2] も存在し，このインターフェイスがデファクトとなりつつある．
- 起動 OS は事前に登録され，Amazon のストレージサービスである S3(Simple Storage Service)[3] に配置されたものでなくてはならない．
- 仮想計算機に割当てられる IP アドレスは動的に定まる．静的なグローバル IP アドレスを特定の仮想計算機に割り振ることも可能だが，別途費用が発生する．
- 仮想計算機のディスクイメージは，終了時に保存されない．すなわち，仮想計算機を一度停止すると停止前にディスクイメージに対して行った変更はすべて失われてしまう．これを避けるためには，変更を S3 に書き出す，もしくは EDS を用いて明示的に不揮発のディスクスペースを割り当てなければならない．

3. GridRPC システム Ninf-G

Ninf-G[4),5)] は RPC(Remote Procedure Call) 機構をグリッド上で実現する GridRPC システムである．GridRPC[6] は，OGF(Open Grid Forum) で標準化が進められている API 規格で，Ninf-G はこの API 規格に準拠している．

Ninf-G は大きく分けてクライアントとリモート計算モジュールの二つのプログラムから構成される．クライアントは，サーバ上のリモート計算モジュールに計算を依頼し，結果を受け取る．ひとつのクライアントから，複数のリモート計算モジュールを同時に利用することで，並列実行を容易に実現することができる．

3.1 クライアントプログラム

Ninf-G のクライアントプログラムの例を図 1 に示す．ハンドルと呼ばれる構造を，サーバと実行プログラム ID を指定して作成し，それに対して `grpc_call_async` で引数を指定して計算を依頼する．ユーザが引数のマーシャリングを明示的に行う必要がないのが，GridRPC の特徴である．また，呼び出しは非同期に行われるため，複数の呼び出しを並行して行うことで容易に並列実行を実現することができる．

リモートサイト上に計算モジュールを起動はハンドル作成時に行われる．一度作成したハンドルに対して複数回の呼び出しを行うことが可能である．従って，各呼び出し時には起動コストがかからず，軽量に呼び出しを行うことができる．

図 1 に示した例では，複数のハンドルを一括して作成する API(`grpc_function_handle_array_init_np`) を用いている．この API はバックエンドに array job 対応のジョブキューイングシステムを用いる場合には，array job を用いることで，軽量に多数のハンドルを取得することが可能となっている．

Vol.2009-HPC-121 No.22
2009/8/5

```
grpc_function_handle_t handles[N];
grpc_error_t res;

grpc_function_handle_array_init_np(handles, N,
          "serverTag"
   "test/func);
    ...
for (i = 0; i < N; i++) {
  grpc_call_async(&(handles[i], 100, A, B);
}
grpc_wait_all();
```

図 1 クライアントプログラムの例

3.2 起動プロキシと通信プロキシ

Ninf-G5 は，様々なジョブ起動プロトコル，通信レイヤを使用できるよう，ジョブ起動部分，通信管理部分を外部プロセスとして分離する構成を取っている[7]．ジョブ起動を管理するプロセスを起動プロキシ，通信を管理する外部プロセスを通信プロキシと呼ぶ．一般に通信プロキシはクライアント側プロキシとリモート側プロキシの対で構成される．

これらのプロキシとクライアントライブラリは，テキストベースの簡易なプロトコルで通信する．テキストベースとすることで言語独立性を担保している．とくにジョブ起動プロキシの Java 言語実装に関しては通信ライブラリが提供されており，容易に新たなジョブ起動機構を追加できるよう配慮されている．

以下，EC2 で用いる ssh 起動プロキシと ssh 通信プロキシを概説する．

3.3 ssh 起動プロキシ

ssh 起動プロキシは起動プロキシの一つで，ssh を用いてジョブの起動，モニタ，制御を行う．さらにリモート計算モジュールのステージイン，リモート計算モジュールの標準出力，標準エラーのステージアウトにも対応する．

ssh 起動プロキシは，起動時にサーバに対してセッションを確立し，サーバ側にシェルを起動して，そのシェルを経由してジョブ起動，モニタ，制御を行う．ステージイン，ステージアウトはそれぞれ個別の ssh セッションを用いて行う．ここで，scp は用いていない．scp はサーバによってはサポートされていない場合があるためである．

ssh 起動プロキシは，バックエンドとしてキューイングシステムを利用することができる．現在 Sun Grid Engine，Torque, Condor がサポートされている．デフォルトでは fork を用いて，リモートノード上で直接実行する．

3.4 ssh 通信プロキシ

通信プロキシの一つとして，ssh 通信プロキシがある．このプロキシは，ssh で構成される通信路をトンネルとして利用して，クライアント，リモートサイト間の通信を中継する．このプロキシを用いることで，NAT 内部からの Ninf-G プログラムの実行が可能になる．

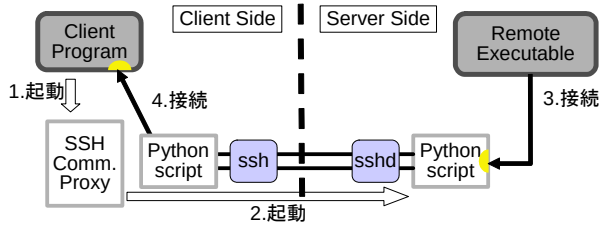

図 2 ssh 通信プロキシ

通常の通信プロキシでは，リモート側の通信プロキシはリモート計算モジュールから起動されるが，ssh 通信プロキシにはリモート側通信プロキシがなく，クライアント側の通信プロキシから ssh 経由で起動される Python のスクリプトがその役を担う．

ssh 通信プロキシの動作の様子を図 2 に示す．ssh 通信プロキシは，ssh が提供する標準入出力路を多重化し，ポートフォワーディングを行う．この機能を実現するために Python のスクリプト ssht を開発した．ssht はリモートサイト上に ssh 経由で Python インタプリタを起動し，自分のコピーを送り込む．リモート側の ssht は，クライアント側の指定に従ってポートの開閉，ストリームのフォワードを行う．

ssht の機能は，ssh の標準機能であるポートフォワーディングと一部重複するが，ssh の標準機能には下記の問題点があるため，ssht を用いた．

- サーバ側の設定でポートフォワードを明示的に許可する必要があり，使用できないサイトがある．
- リモート側のフォワードするポートを ssh 起動時に静的に指定するため，そのポートが利用できなかった場合には ssh を再度行うことになり，リトライのコストが大きい．

図 3 に ssh 起動プロキシと ssh 通信プロキシを併用して，Condor で管理されているクラスタ上でプログラムを実行する際の，Ninf-G の動作概要を示す．クライアントはまず，ssh 通信プロキシを起動し，対象サイトのゲイトウェイノードまでトンネルを構築する．次に ssh 起動プロキシを利用し，リモート実行モジュールを Condor に投入する．Condor は適切なノード上でモジュールを起動する．起動したモジュールは，ssh 通信プロキシの構築したトンネルを経由してクライアントに接続して RPC リクエストを受け付ける．

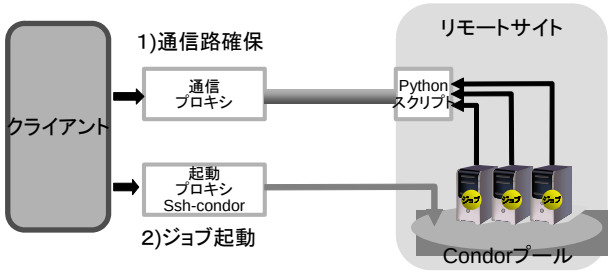

図 3 ssh 起動プロキシと ssh 通信プロキシ

4. 設計と実装

4.1 要　　請

以下に提案システムへの要請を整理する．

(1) クラウド側に対して実行時に設定をする必要がないこと．

(2) 既存のクライアントプログラムに対して変更を加えずに実行できること．特に，大規模なマスタワーカ型のクライアントプログラムがそのまま稼働すること．Ninf-G5 の大規模マスターワーカプログラムは，バックエンドにジョブキューイングシステムを仮定し，上述の複数ハンドル一括作成 API を用いるので，これがそのまま稼働することが望ましい．

4.2 設計の指針

Ninf-G5 で EC2 上のリソースを利用する方法は大きく分けて 3 つ考えられる．

(1) 別の枠組みで，EC2 にインスタンス群を作成し，そのホスト名を個別に埋め込んだ設定ファイルを生成して，Ninf-G5 を実行する．起動プロキシとしては通常の ssh を用いる．

(2) 起動プロキシで EC2 の制御とジョブ起動の制御を行う．

(3) EC2 上に複数のノードから構成されるバッチキューイングシステムをデプロイし，そこに ssh 起動プロキシを用いてジョブを投入する．

指針 (1) に基づいた実装では，個々のノードを個別に呼び出すことになり，複数ハンドル一括作成 API を利用することができない．このため要請 (2) を満たすことができない．一方指針 (3) に基づくと，要請 (2) は満たすことはできるが，EC2 上での大掛かりなデプロイが必要になり要請 (1) を満たさない．したがって，我々は指針 (2) を採用した．

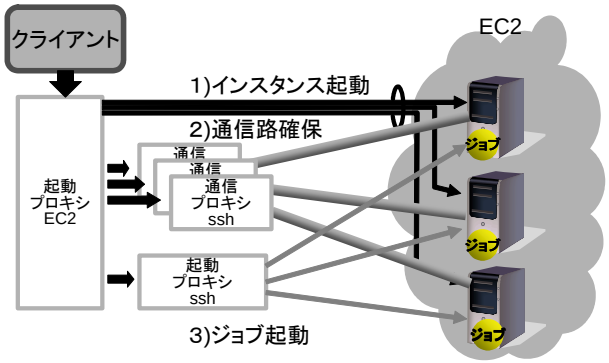

図 4 EC2 起動プロキシを用いた Ninf-G の動作

4.3 EC2 起動プロキシの設計

EC2 起動プロキシは，1)EC2 上のインスタンスの管理，2) インスタンス上でのジョブ起動，管理，3) インスタンスへの通信路の確保，の 3 つの役割を担う．このうち 2) の機能は ssh 起動プロキシと，3) の機能は ssh 通信プロキシと重なる．このため，これらの機能に関しては，これらのモジュールをカスケードして呼び出すことで実現する．図 4 に概念図を示す．

EC2 起動プロキシは，起動時に設定ファイルで指定されたノード数のインスタンスを EC2 上に確保する．以降のリクエストに対しては，ラウンドロビンでノードを割り当てる．

4.4 EC2 起動プロキシの実装

EC2 起動プロキシは，Java で記述した．EC2 の制御には，Amazon EC2 の提供する Java のライブラリを用いた．

Ninf-G の提供する起動プロキシ用の Java ライブラリを利用することで，少量のコードで機能を実現することができた．EC2 起動プロキシのコード規模は 10 ファイル 1600 行程度である．

表 1 に EC2 起動プロキシの主要なオプションを示す．

4.5 動作のサマリ

EC2 を用いた Ninf-G プログラムの動作は以下にまとめる．

(1) ユーザがクライアントプログラムを起動

(2) クライアントプログラムの実行が，EC2 を指定した関数ハンドル作成に行き当たっ

表 1 EC2 起動プロキシの主要オプション

名前	意味
EC2_numnodes	EC2 上で使用するノードの数
EC2_accessKeyId	EC2 アクセスキー
EC2_secretAccessKey	EC2 シークレットキー
EC2_imageId	OS イメージ ID
EC2_keyName	SSH キーの登録名
EC2_useSshCommProxy	通信プロキシを利用するかどうか
EC2_keyFile	SSH 秘密鍵ファイル名
EC2_sshUser	ログインユーザ名
EC2_subInvokeServerLog	SSH 起動プロキシのログファイル名
EC2_subCommProxyLog	SSH 通信プロキシのログファイル名

た時点で EC2 起動プロキシが起動.

(a) EC2 起動プロキシは，設定で指定された個数のインスタンスを EC2 上にプロビジョン.

(b) ssh ログインが可能になるのを確認し，すべてのインスタンスに対して，ssh 通信プロキシを起動.

(c) ssh 起動プロキシを 1 つだけ起動して，依頼された関数ハンドルを作成.

(3) 以降の関数ハンドル作成リクエストは，EC2 起動プロキシから，ssh 起動プロキシに委譲.

(4) ユーザプログラム実行終了時，EC2 起動プロキシに EXIT コマンド送出．EC2 起動プロキシはプロビジョンしたインスタンスをすべて停止して終了.

5. 評　　価

5.1 評価環境

評価環境を図 5 に示す．クライアントとしては産総研秋葉原サイトに設置した PC を使用した．CPU は AMD Athlon 64 X2 3800+，OS は Linux 2.6.28-13 i686 である．

EC2 サイトとしては us-east1 を利用した．インスタンスの OS イメージにはすべて amazon の提供する 32bit の fedora 8 イメージを用いている．

また，対照試験用にローカルサイトにもサーバを用意した．ローカルサーバは Intel Core 2 Quad 9550 で，OS は Linux 2.6.18 x86_64 である．ローカルサーバとの間のネットワークは 100base/TX で iperf による測定では上り下りとも 94Mbps 程度，ping レイテンシは 0.3ms 程度である.

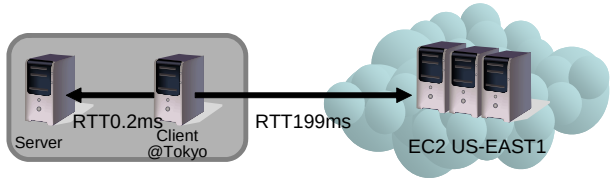

図 5 評価環境

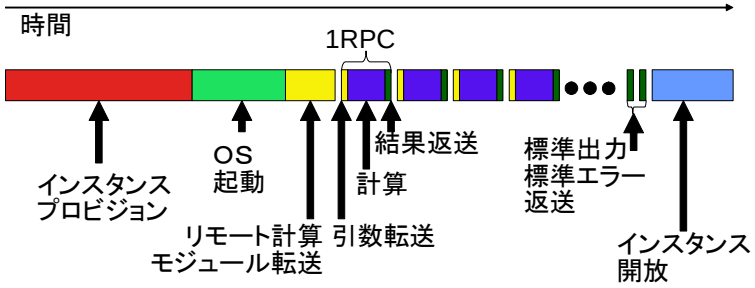

図 6 動作のタイムライン

5.2 動作のタイムライン

EC2 を用いたマスタワーカプログラムが動作する際の一つのインスタンスの動作タイムラインを図 6 に示す．まず，EC2 上にインスタンスが確保され，その上で OS が起動する．つぎに，ワーカプログラムに相当するリモート計算モジュールが ssh 経由でステージされ，起動される．リモート計算モジュールはクライアントと通信し，複数回の RPC 呼び出しを処理する．一度の RPC 呼び出しは，引数の送信，計算，結果の返送から構成される．すべての呼び出しが終了し，クライアントプログラムが関数ハンドルを破棄すると，クライアントライブラリは，リモート計算モジュールに標準出力と標準エラーを返送させ，リモート計算モジュールを停止する．最後に，EC2 インスタンスを終了する.

このうち，インスタンスの確保と OS の起動にかかる時間に関しては，次節で論じる．リモート計算モジュールのステージングにかかる時間は，約 2Mbyte のプログラムの 9 秒程度であった．標準出力と標準エラーの転送には，サイズ 0 であってもそれぞれ 3 秒程度を要した．これは，長大なレイテンシのため，ssh の認証に時間がかかるためである．EC2 インスタンスの終了にはおよそ 21 秒程度の時間がかかる.

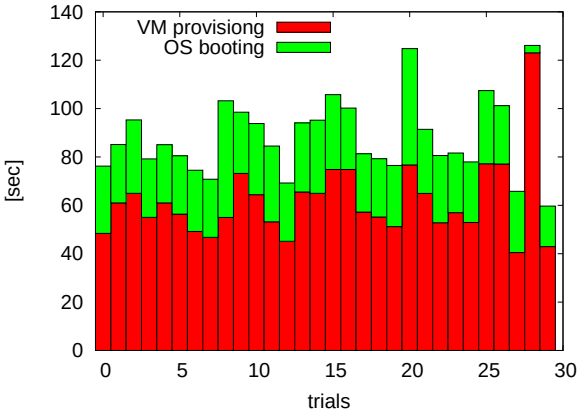

図 7 EC2 上での OS プロビジョニング時間

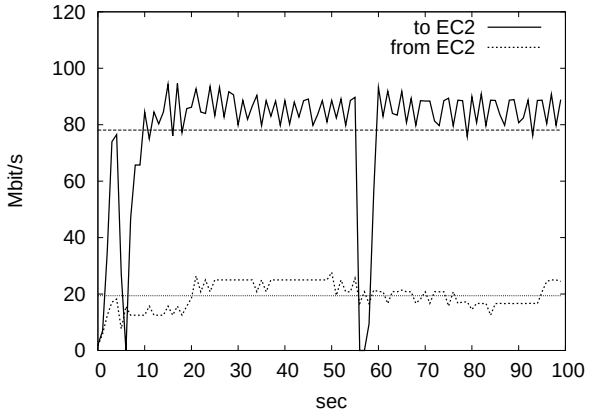

図 8 EC2 インスタンスとの間のスループット

5.3 プロビジョニング時間

EC2 上でジョブを実行するまでには，1) 仮想計算機が EC2 上でプロビジョンされ，2) 用意された仮想計算機上で OS がブートする，まで待たなければならない．

これらにかかる時間を計測した．計測は 30 回連続して行った．仮想計算機のプロビジョニング完了は，EC2 インスタンスの状態が”running” になった時点で終了と判定し，OS ブート完了は ssh でログインが可能になったことをもって判定した．これらの判定は 1 秒ごとのポーリングで行っている．計測結果を図 7 に示す．

仮想計算機プロビジョニング時間 (図中赤) の平均は 61 秒，OS ブートまでを含めた平均は 88 秒，偏差は 15.5 秒であった．最後から 2 回目の試行の際には，プロビジョニング時間が異常に長く，OS ブート時間が異常に短くなっているが，これは不安定なネットワークのために，EC2 インスタンスの状態遷移の把握が遅れたためであると思われる．

5.4 EC2 との通信速度

EC2 上のインスタンスとの通信速度を測定した．測定には iperf[8] を用い，メッセージサイズは iperf の自動調整に任せた．測定は，EC2 側をサーバにした場合，東京側をサーバにした場合の双方で行い，1 秒間隔で 100 秒間測定した．結果を図 8 に示す．100 秒間の平均は上りが 78.1Mbit/s, 下りが 19.4Mbit/s であった．

上りは比較的高スループットではあるが，頻繁に途絶することがあることがわかる (図中 6 秒，58 秒近辺), 下りは上りに対して 4 分の 1 程度しか出ていない．

また，EC2 側から ping で測定したレイテンシは，199ms であった．

5.5 Ninf-G5 の通信速度

Ninf-G5 を利用した場合の実スループットを計測した．図 9 に結果を示す．横軸はデータサイズ，縦軸はスループットである．通信プロキシを用いた場合と用いない場合，上りと下りをそれぞれプロットした．参考として，ローカルサイトで実験した結果を図 10 に示す．

EC2 では上り下りとも，データサイズの増大に対するスループットの立ち上がりが遅い．また．通信プロキシを用いた場合の性能低下が大きい．特に通信プロキシを用いた場合には，上りと下りのスループットが逆転していることが目を引く．

5.6 議　　論

EC2 を用いたマスタワーカ計算の実用性を考えるために，問題とノード数を仮定して，コストを計算してみる．

10000 のタスクに分割できるジョブを考える．各タスクの入出力は 100Kbyte，各タスクは，現代の標準的な PC でスカラ実行した場合に 3 秒かかると仮定する．すなわち，このジョブをシリアルに PC で実行すると 3 0000 秒かかることになる．ワーカとなるリモート計算モジュールのサイズは 2Mbyte とする．

これを EC2 を利用して実行する場合の実行時間とコストを計算してみる．まずノード数に関わらず一定である部分の時間を考える．図 6 中，RPC 部以外がそれに該当する．インスタンスプロビジョニングと OS 起動で 88 秒，リモート計算モジュールの転送に 9 秒，標

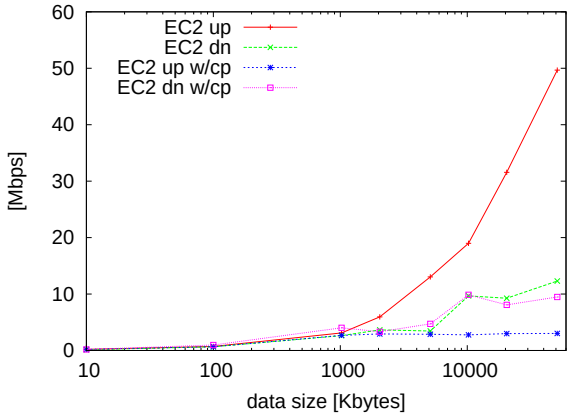

図 9 Ninf-G のスループット (EC2)

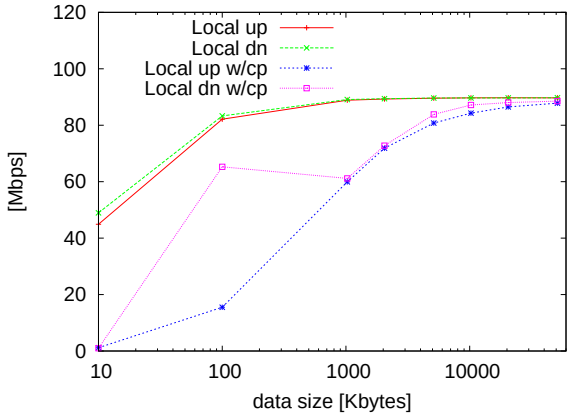

図 10 Ninf-G のスループット (ローカルサイト)

準出力，エラーの返送にそれぞれ 3 秒，インスタンスの終了で 21 秒，総計で 124 秒である．

各タスクの実行には，入出力の通信時間と計算時間がかかる．EC2 のインスタンスの計算ユニットは，現代の標準的な PC と比べると低速なので，計算速度の比を r と置くと，各タスクの実行には $3/r$ 秒かかることになる．100Kbyte のデータの送受信には，いずれも約 1 秒かかることから，各タスクの実行時間は $2+3/r$ 秒となる．10000 ジョブを完全に並列

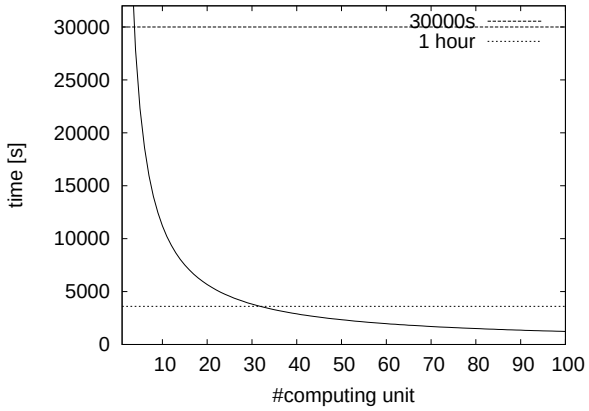

図 11 使用計算ユニットと実行時間

実行できると仮定すると，N ノードに対して $floor(10000/N)*(2+3/r)$ 秒ですべてのタスクの処理が可能である．ここで floor は引数を超えない最大の整数を返す関数である．

EC2 の計算ユニットが標準的な PC の 1/3 の速度であると仮定し，r=0.33 とした結果を図 11 にしめす．100 計算ユニットで 1233 秒，40 計算ユニットで 2896 秒となる．100 計算ユニットでおよそ 24 倍のスピードアップが得られる．

次に課金を考えてみる．EC2 にはいくつかのインスタンスタイプがあるが，計算ユニット当りに対する課金が最も安い High CPU を仮定すると，5 計算ユニットで 1 時間あたり 0.20 ドルである．課金は 1 時間単位で切り上げられる．20 インスタンス 100 計算ユニットを利用した場合，4 ドルとなり，8 インスタンス 40 計算ユニットの場合には 1.6 ドルとなる．4 インスタンス 20 計算ユニットの場合には，実行時間が 5669 秒と 1 時間を超えるため，2 時間分でやはり 1.6 ドルとなる．

通信量は，上り下りともに 1Gbyte 程度なので，0.1 ドル+0.17 ドルで 0.27 ドルである．これらをプロットしたものを図 12 に示す．基本的に計算ユニット数の増大に従ってコストが増大するが，30 計算ユニット以下では，実行時間が 1 時間を超えるため，価格が大きく変動している．100 計算ユニット使用した場合，24 倍で 4 ドル 27 セントと，比較的安価にスピードアップが得られることがわかる．

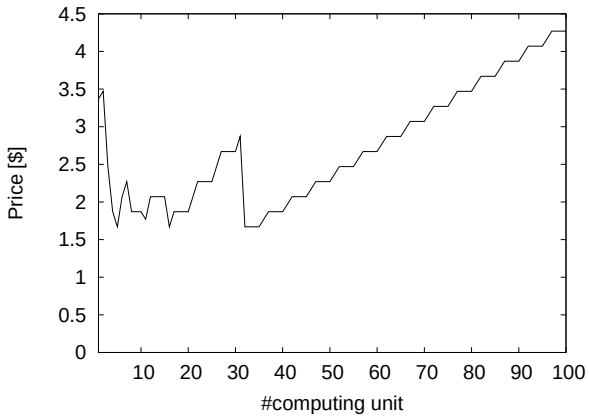

図 12 使用計算ユニットと課金

6. 関 連 研 究

Condor[9] は，RedHat Enterprise Linux の一部としても配布されているが，この機能の一つとして，EC2 を制御するモジュールが提供されている[10]．オンデマンドで EC2 上に簡易ジョブスケジューラを構築し，ここに Condor に投入されたジョブを投入することでユーザからはシームレスに EC2 が利用できる．この機構は 4.2 で列挙した指針のうち (3) に近い．EC2 上に簡易ジョブスケジューラを構築する必要があり，したがってイメージを S3 に維持する必要がある．

昨今，Surge Computing という概念が提唱されている[11]．これは，通常の計算は小規模なインハウスの計算リソースで行い，まれに生じる高負荷時のみにクラウドなどの外部リソースをシームレスに利用することで，トータルコストを低減する手法である．これまでも Ninf-G5 では，ローカルな計算資源，SGE，TORQUE をもちいたクラスタ環境，Globus や Condor を用いた大規模なグリッドをシームレスに併用することを実現していたが，本稿で提案した機構を用いることで，これらに加えて，EC2 のクラウド環境をもシームレスに利用することが可能となっている．

7. お わ り に

HaaS 型クラウド上でのマスタワーカ型科学技術計算アプリケーション実行の可能性を確認するために，GridRPC システム Ninf-G5 の起動プロキシを実装することで，代表的な HaaS 型クラウドである EC2 でワーカプロセスを実行する分散計算機構を実現した．簡単な評価を行った結果，起動のコスト，通信レイテンシが大きく，通信スループットが低いという問題はあるものの，計算量の大きいアプリケーションでは十分な性能向上が得られることがわかった．また，この機構はクラウド側には ssh ログイン以外の機能を要求しないので，他の HaaS 型クラウドにも容易に転用が可能であると思われる．

今後は，大規模な実アプリケーションでの評価を行い，本機構の有効性を確認していく予定である．

参 考 文 献

1) Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/.
2) Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L. and Zagorodnov, D.: The Eucalyptus Open-source Cloud-computing System, *Proceedings of 9th IEEE International Symposium on Cluster Computing and the Grid* (2009).
3) Amazon Simple Storage Service (Amazon S3), http://aws.amazon.com/s3/.
4) Nakada, H., Tanaka, Y., Matsuoka, S. and Sekiguchi, S.: *Grid Computing: Making the Global Infrastructure a Reality*, chapterNinf-G: a GridRPC system on the Globus toolkit, pp.625–638, John Wiley & Sons Ltd (2003).
5) 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).
6) Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H.: GridRPC: A Remote Procedure Call API for Grid Computing, submitted to Grid2002.
7) Nakada, H., Tanimura, Y., Tanaka, Y. and Sekiguchi, S.: "The Proxy-based Design Pattern for Grid Middleware and its Application: Ninf-G5", *"Proceedings of HPC ASIA 2009"*, pp.210–217 (2009).
8) Iperf. http://sourceforge.net/projects/iperf.
9) Condor. http://www.cs.wisc.edu/condor/.
10) Farellee, M.: Red Hat and Condor Project Collaboration. http://www.cs.wisc.edu/condor/CondorWeek2009/condor_presentations/farrellee-redhat-condor.pdf.
11) Avobe the Cloud: Surge Computing/Hybrid Computing. http://berkeleyclouds.blogspot.com/2009/05/surge-computing.html.