グローバルコンピューティングのための スケジューリングフレームワーク

中田秀基[†1] 竹房あつ子[†2] 松岡聡[†3,†4]
佐藤三久[†5] 関口智嗣[†1]

ネットワーク技術の発展にともない，グローバルコンピューティングシステムがいくつか提案されているが，その計算/通信リソースを十分活用するための手法はいまだ確立されていない．リソースを有効に活用するためには，グローバルコンピューティングに特有なリソースの変動と不安定さを考慮し，関連する複数のタスクに対して一括して適切にリソースを割り当てるスケジューリング手法が必要である．本稿では，グローバルコンピューティング環境において個々のアプリケーションの性能とシステム全体のスループットを両立させるための，階層化されたスケジューリングフレームワークを提案する．さらに，このフレームワークに準拠した Ninf システムのメタサーバスケジューリングフレームワークの実装について述べ，試験的に複数のスケジューリング手法をメタサーバ上に実装しその動作を確認する．この結果，本フレームワークが十分にフレキシブルであることが示された．

A Scheduling Framework for Global Computing

Hidemoto Nakada,[†1] Atsuko Takefusa,[†2]
Satoshi Matsuoka,[†3,†4] Mitsuhisa Sato[†5]
and Satoshi Sekiguchi[†1]

Rapid progress in networking technology is now making global computing systems feasible. Although there have been proposals of global computing systems, it is still a research issue as to how to achieve efficient usage of computing resources in global computing. In particular, we need to devise appropriate scheduling strategies/algorithms of computing resources over wide-area networks, which are often dynamic and unstable in nature. This paper presents our preliminary scheduling framework for unifying application and job scheduling in global computing. The proposed framework establishes a layer of scheduling and resource allocation subframeworks. We show our software framework Ninf metaserver which provides low-level scheduler and resource monitor. We also evaluate some scheduling strategies using the framework. The evaluation results prove that the framework is flexible enough to implement plural scheduling algorithms on top of it.

1. はじめに

ネットワーク技術の発展により，広域ネットワーク上に分散した計算リソースや情報リソースを積極的に活用し大規模計算を実現するグローバルコンピューティングが可能となり，これを目的としたシステムが Ninf[1] をはじめ，複数提案されている[2)~4)]．しかしグローバルコンピューティングシステムの計算/通信リソースを十分活用するための手法はいまだ確立されていない．リソースを有効に活用するためには，広域分散計算に特有なリソースの変動と不安定さを考慮し，関連する複数のタスクに対して適切にリソースを一括して割り当てるスケジューリング手法が必要である．

従来のグローバルコンピューティングシステムでのスケジューリングの研究は，**アプリケーションスケジューリング**と**ジョブスケジューリング**の 2 つに大別することができる．アプリケーションスケジューリングは，単一のプログラムの応答時間を短縮することを目的とし，アプリケーションプログラムの特性に従っ

†1 電子技術総合研究所
Electrotechnical Laboratory
†2 お茶の水女子大学
Ochanomizu University
†3 東京工業大学
Tokyo Institute of Technology
†4 科学技術振興事業団
Japan Science and Technology Corporation
†5 新情報処理開発機構
Real World Computing Partnership

てプログラムを複数の並列タスクに分割してタスクをスケジュールする．これに対してジョブスケジューリングはグローバルコンピューティングシステム全体に投入した複数ジョブの総実行時間（スループット）を向上させることを目的として，各ジョブをスケジュールする．

アプリケーションスケジューリングの例としてはAppLeS [5]，Prophet [6]，Globus [4]のDUROC [7]などがある．これらのシステムはいずれも1つのクライアントプログラムがそのグローバルコンピューティングシステムを占有する状態を前提としており，複数のクライアントのプログラムがグローバルコンピューティングシステムを共有し，お互いのタスクが干渉することは考慮されていない．Netsolve [2]のAgentは，クライアントに適切なサーバを紹介する，ある種のアプリケーションスケジューリングシステムであるが，Netsolveの性質上単純なRPCの構造に対するスケジューリングのみを対象としており一般性に欠ける．

ジョブスケジューリングはMPPのように密に結合した環境では広く研究されてきた[8]が，計算/通信リソースが不安定に変動するグローバルコンピューティングシステムに関しては十分研究されていない．Condor [9]のMatchmakingメカニズムは計算/通信リソースとそれに対する要求を構造化された言語で記述し，リソースと要求の対応をとることによってリソースの割当てを行うものである．これは，複数のジョブをそれぞれ適切なプロセッシングユニットにディスパッチするという意味においては一種のジョブスケジューラといえる．しかし，その主眼が特定の資質（アーキテクチャ/デバイスなど）を持つリソースを確保することにおかれており，複数のジョブ全体のスループット向上をめざすものではなく，狭義のジョブスケジューラには該当しない．また，計算/通信リソースの負荷などの動的情報，複数のジョブ間の干渉，自分自身のサブタスク間の干渉は考慮されていない．

パラメータサーチに特化したグローバルコンピューティングシステムNimrod [10]は，スループット指向のジョブスケジューラを持つが，そのアルゴリズムは単純なセルフスケジューリングにすぎない．

ここで重要なのは，従来のグローバルコンピューティングにおける研究ではアプリケーションスケジューリングとジョブスケジューリングが協調して論じられていないということである．これは，これまでのグローバルコンピューティングシステムの研究が事実上単一のジョブ/アプリケーションをターゲットにした専用システムとして開発されてきたことに起因すると思われる．これまでは，グローバルコンピューティングシステムが未成熟だったこともあり，個々のシステム上では対象とするジョブ/アプリケーションを個別に動かすことしかできず，それに特化したスケジューリング機構で十分であった．フレームワークが成熟しつつある現在，アプリケーションスケジューリング的な観点とジョブスケジューリング的な観点をバランス良く組み合わせ，個々のアプリケーションの性能とシステムの総スループットを両立させるスケジューリングが求められている．これを実現するには，複数のジョブ間，サブタスク間の干渉を考慮にいれたうえで，タスク全体を一括でスケジューリングするスケジューリングアルゴリズムとソフトウェアフレームワークが必要である．

本稿では，グローバルコンピューティングにおいてアプリケーションスケジューリングとジョブスケジューリングを統合するための基本的なスケジューリングフレームワークについて述べる．また，そのフレームワークの一部である低位レベルスケジューリングシステムの実装例としてNinfシステムのメタサーバスケジューリングフレームワークの実装を示す．さらに，メタサーバフレームワーク上に複数の試験的なスケジューラを実装し，簡単なアプリケーションを用いて評価する．この結果，本フレームワーク上で複数のスケジューリングアルゴリズムを実装，比較検討できることから，十分なフレキシビリティを持つことが確認できた．

以後，2章でグローバルコンピューティングでのスケジューリングフレームワークについて述べ，3章でNinfメタサーバフレームワークについて説明する．4章では，いくつかのスケジューリング手法について，メタサーバのプロトタイプを用いた実環境での評価について報告する．5章でまとめおよび今後の課題について述べる．

2. グローバルコンピューティングスケジューリングフレームワーク

2.1 グローバルコンピューティングシステム

本稿で対象とするグローバルコンピューティングシステムは一般に以下の要素で構成される．

- クライアント：計算要求を発行する主体
- 計算サーバ：計算リソースを提供するサーバ
- ストレージサーバ：計算の途中結果や共有されるべきデータを保持するサーバ
- スケジューラ：計算要求に対して計算サーバやストレージサーバを割り当てる機構

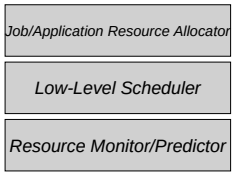

図 1 グローバルコンピューティングスケジューリングフレームワーク
Fig. 1 Scheduling framework for global computing.

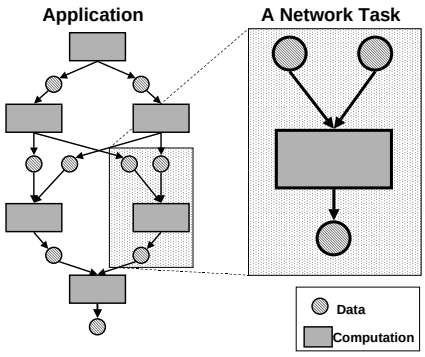

図 2 Network Task
Fig. 2 Network Task.

2.2 スケジューリングフレームワークの概要

単一ジョブの実行性能と複数ジョブの総合スループットを両立させるには，それぞれのジョブを粗粒度のタスクに分割して，それを単一の枠組みでスケジュールすればよい．このための枠組みとして我々は**図 1** に示すような階層化したスケジューリングとリソース割当てのサブフレームワークを提案する．

本フレームワークのポイントは，ジョブ/アプリケーションを，Network Task と呼ぶ，粒度の大きいタスクに分割してスケジュールすることにある．Network Task は，アプリケーションを構成する計算を大粒度に分割した計算単位であり，計算とそれに対する入力データ，出力データで定義される．アプリケーションは，Network Task をノードとする一種のマクロデータフローグラフとして表現することができる（**図 2**）．

本フレームワークでは，まず，ジョブ/アプリケーションリソースアロケータがアプリケーションの通信/計算特性を考慮しつつ Network Task グラフへの分割を行う．次に，低レベルスケジューラではサーバやネットワークの現在の負荷や予測された負荷情報を考慮し，個々の Network Task を適切な計算ノードに割り当てる．この際に，Network Task の構造に反映されているアプリケーションの構造を利用して，個々のアプリケーションに特化した低レベルスケジューリングを行う．また，最下位のリソースモニタ/プレディクタは低レベルスケジューラにスケジューリングに必要となるリソースの状況に関する情報を提供する．

このスケジューリングフレームワークで最も着目すべき点は，高レベルのアプリケーションスケジューリングとジョブスケジューリングの問題を低レベルの Network Task に統一するところにある（Network Task スケジューリング）．低レベルスケジューラは Network Task のスケジューリングだけを行うのだが，結果としてアプリケーションスケジューリングとジョブスケジューリングの要求がバランス良く取り入れられたスケジューリングが実現される．また，Network Task をベースとして考えることでグローバルコンピューティングの基本性能モデルが単純になり，計算サーバやネットワークの負荷情報だけを，スケジューリング決定のためのパラメータとすることができる．これによって，動的な環境の変化へ追従したスケジューリングが容易になり，グローバルコンピューティングプラットフォーム固有の不安定な環境下でのスケジューリングが可能になる．

2.3 Network Task への分割

Network Task は，状態を持たない大粒度の計算単位である．個々の Network Task は一般に複数の入力データを受け取り，それに対して演算処理を行い，複数の出力データを出力する．Network Task の属性はそのタスクでの計算量，および上流，下流の Network Task との依存関係と通信量で規定される．

Network Task の粒度は，上位スケジューラによって決定される．この際に重要なのは，ネットワーク経由の並列実行による効果が得られるよう，通信量と計算量の比率が適切になるように，Network Task を決定することである．このために，上位スケジューラは各 Network Task の計算量，および通信量を取得し，これらの情報を用いてプログラムを分割する．計算量/通信量を取得する方法としては，プログラムから直接抽出する方法，プログラマがプラグマでヒントを提供する方法が考えられる．

このような分割は，科学技術分野における大規模計算では比較的容易に行うことができる．たとえば，パラメータを変動させて各パラメータ値に対応する値を計算する，パラメータサーチと呼ばれる型の演算は，各パラメータに対する演算部をタスクとして切り出すことで，ネットワークタスクに分割できる．また，たんぱく質の類似検索による構造決定なども有望なアプリケーションである．

2.4 Network Task スケジューリング

低レベルスケジューラは Network Task グラフを個々の Network Task の計算量，Network Task 間の通信量と，計算/通信リソースの状況を勘案し，Network Task グラフをリソースにマップする役割を持つ．この際問題となるのは，過去にスケジュールされた実行中の Network Task 群や，同一グラフ内の他の Network Task との間に生じる干渉である．広域ネットワーク環境下では，計算/通信リソースのモニタリングは，そのコストや計算への影響を考慮すると，高い頻度で行うことは現実的ではない．さらに，モニタした情報をスケジューラへ転送することにも時間がかかるため，収集されたリソース状況はつねに過去のものであることになり，これをスケジューリングの基礎情報としてそのまま用いることはできない．このため，適切なスケジューリングを行うには，収集された情報に基づいて現在の状況を予測する必要がある．さらに，同一グラフ内の Network Task の配置による突発的なリソースの状況変化を知ることは，負荷の過度の集中を避けるために非常に重要であり，配置の記録に基づいてこれを予測する必要がある．このために，広域に分散した計算・通信リソースの情報を収集し集中的に管理する DB マネージャとそれを参照してリソース状況の予測を行うプレディクタを導入する．

低レベルスケジューリングのためのサブフレームワークは以下のモジュールから構成される．

- 計算リソースの負荷モニタ
- ネットワークリソースのモニタ
- リソース情報 DB マネージャ
- リソース状況プレディクタ
- リソーススケジューラ

負荷モニタは計算サーバの情報を収集する．ネットワークリソースモニタは，システムを構成するすべてのサーバ/クライアント間のスループットを測定する．これらのモニタは定期的に観測を行い，その結果をリソース情報 DB マネージャに報告する．リソース状況プレディクタは DB に格納された情報に基づいて，そのリソースの現在と未来の状況を予測する．スケジューラは DB とプレディクタの情報をふまえて，各 Network Task に対して適切な計算ノードを割り当てる．このようにリソースの監視と予測を行うことによって，グローバルコンピューティング特有の不安定な環境下での安定したスケジューリングが可能になる．

3. Ninf メタサーバシステムによる試験実装

Ninf システムは，RPC ベースのグローバルコンピューティングシステムである．Network Task スケジューリングを行うフレームワークとして，2.4 節に基づく Ninf メタサーバスケジューリングフレームワークのプロトタイプを設計・実装した．

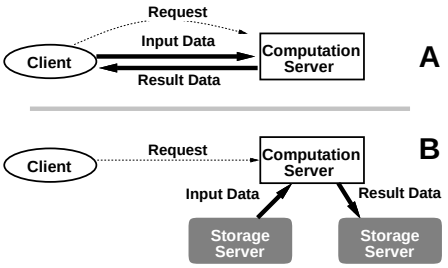

図 3 Ninf サーバとストレージサーバ
Fig. 3 Ninf server and strage server.

3.1 Ninf システム

Ninf システムは基本的にリモートサーバ上の計算ルーチンをクライアントから実行するシステムである．通常の RPC では，計算に用いるデータはすべてクライアントが供給し，結果もクライアントが読み戻す（**図 3** A）．Ninf には，この通常の RPC に加えて，計算に用いるデータを任意のストレージサーバから供給し，計算結果も任意のストレージサーバに書き込ませる機能がある（図 3 B）．この機能を用いて Network Task グラフを Ninf サーバ群にマップして実行することができる．すなわち，ストレージサーバを経由して Ninf サーバ群が協調して動作するのである．ストレージサーバの機能は，現在 Ninf サーバ自身に実装されており，すべての Ninf サーバがストレージサーバとしての役割を果たすことができる．

3.2 メタサーバ

Ninf のメタサーバシステムは，2.4 節に基づいて設計されている．効率的な実装を実現するために，複数の理論的なモジュールが 1 つの物理的モジュールにマップされている部分もあるが，本質的には等価である．

メタサーバは以下のモジュールによって構成される．

- 計算リソースの負荷モニタ
- クライアントプロキシ
- リソース情報 DB マネージャ
- リソース状況プレディクタ
- リソーススケジューラ

2.4 節ではクライアントがスケジューラと直接交信してタスクの配置を問い合わせ，タスクの実行を行うことを想定しているが，Ninf システムではクライアントプロキシがこの役割を担っている．これは，Ninf ではさまざまな言語/システムにクライアントを実装

することを目的の 1 つとしており，クライアントプログラムに複雑な機能を実装することは避けたいからである．

クライアントプロキシの役割で最も重要なことは，クライアントの代理としてリソーススケジューラと通信してスケジュールを実行することであるが，同時に各サーバとの間のネットワークのモニタリングも行う．サイト内すべてのクライアントからネットワークをモニタするかわりにクライアントプロキシが代表してモニタすることでモニタのコスト低減を図っている．さらに，クライアントプロキシからのネットワーク情報のリソース情報 DB への登録は，実際にスケジューリングを依頼する時点で行う．これにより，登録のコストが削減される．

サーバ間のネットワークの状況は各 Ninf サーバ自身がモニタし，リソース情報 DB マネージャに報告する．2.4 節におけるネットワークリソースモニタの機能は，クライアントプロキシと各 Ninf サーバに分散して実装されていることになる．

リソース状況プレディクタには，それまでのスケジューリングの履歴が記録されており，この履歴とリソース DB に登録されている情報を用いて将来におけるリソース状況を予測する．たとえば，すでにタスクを配置したリソースは近い将来負荷が増大するだろう，といった予測が可能である．

メタサーバシステムは，次のように処理を行う（**図 4**）．

(0) Ninf サーバは起動時に，自らの性能をベンチマークプログラムを用いて測定し，実装メモリ量/CPU 数などの情報とともにリソース情報 DB に登録する．

(1) ロードモニタは定期的に計算サーバの負荷情報をモニタし，それをリソース情報 DB に格納する．同様に Ninf サーバは Ninf サーバ間のネットワークをモニタし，リソース情報 DB に格納する．

(2) クライアントプロキシはクライアントプロキシと各サーバ間のネットワークの混雑度などの通信情報をモニタし，その結果をクライアントプロキシで管理する．
同様に，各サーバは互いの間のネットワーク情報をモニタし，定期的にリソース情報 DB に報告する．

(3) 計算要求は Network Task のグラフとして作成される．クライアントはクライアントプロキシに Network Task のグラフを一括して送出する．

(4) クライアントプロキシは受け取った Network Task グラフに対するスケジュール要求をスケジューラに発行する．その際，クライアントプロキシとサーバ間の通信情報も通知する．

(5) スケジューラはリソース情報 DB からリソース状況プレディクタを通して各計算サーバの負荷情報，サーバ間の通信情報を取得する．その情報に基づいて，Network Task のグラフをサーバ群にマップする．
マップした結果はクライアントプロキシに送り返される．同時に，次の予測の際のヒントとしてプレディクタにも送られる．

(6) クライアントプロキシは，スケジューラから受け取ったスケジュール情報に基づいてそれぞれの Network Task を個々のサーバに発行する．
この際，Network Task の依存関係に基づいて Network Task の発行順序をスケジューリングする．さらに，すべての Network Task の実行終了後，ストレージサーバに残った中間結果を削除する．

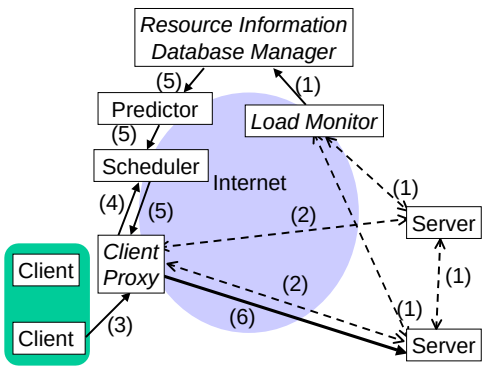

図 4 Ninf メタサーバアーキテクチャ
Fig. 4 Ninf MetaServer architecture.

メタサーバフレームワークのモジュール群は Java で実装されている．スケジューラ/プレディクタモジュールのアルゴリズム部分には，SPI（Service Provider Interface）が定義されており，この SPI を満たした独立したクラスとして，アルゴリズムを記述するよう設計されている．独自のアルゴリズムを用いるには，アルゴリズムを記述した SPI を満足するクラスを記述し，コンフィギュレーションファイルの中のクラス名を置き換えるだけでよい．このように明示的なインタフェースを設けることにより，スケジューリングアルゴリズムの変更が容易になっている．

4. スケジューリング手法の予備的評価

本スケジューリングフレームワークの有効性を確認

するために，メタサーバ上に簡単なスケジューリング手法を複数実装するとともに，サンプルプログラムを実行して実行時間などを測定した．

4.1 Network Task グラフの記述

現在は Network Task グラフを生成する上位スケジューラが未実装であるため，クライアントプログラムからの計算要求として Network Task のグラフを直接記述しなければならない．これには，Ninf の transaction 機能[11]を用いた．transaction 機能はクライアントプログラム上で複数の Ninf 呼び出しを指定し，それらを一括して発行する機能である．クライアントライブラリが，Ninf 呼び出し間のデータ依存関係を自動的に検出して Network Task グラフの作成を行う．

下の例は行列 A，B，C，D を掛け合わせるコードである．

```
Ninf_transaction_begin();
Ninf_call("mmul", N, A, B, E);
Ninf_call("mmul", N, C, D, F);
Ninf_call("mmul", N, E, F, G);
Ninf_transaction_end();
```

各 Ninf 呼び出し “mmul” の第 1 引数 N が行列サイズを，第 2，第 3 引数が入力行列を，第 4 引数が出力行列を示している．この場合**図 5** のような Network Task グラフが生成される．

Network Task スケジューリングに必要となる各 Network Task の計算量，Network Task 間の通信量は Ninf のクライアントライブラリが算出する．Ninf 呼び出しの対象となる関数は，Ninf IDL と呼ばれるスクリプトを用いてその性質が定義されている．IDL には，各引数の入出力モード，サイズ，その関数の計算のオーダを記述する．例として下に上記 “mmul” の IDL スクリプトを示す．

```
1: Define mmul(
2:          IN  int n,
3:          IN  double A[n][n],
4:          IN  double B[n][n],
5:          OUT double C[n][n])
6: CalcOrder n^3
7: Calls "C" mmul(n,A,B,C);
```

3〜5 行めで，各引数の配列は各次元のサイズが第 1 引数 “n” の 2 次元配列であると定義している．したがって配列のサイズは，“n” の 2 乗であることが分かる．また，6 行めの CalcOrder 文で，この関数の計算量が第 1 引数 “n” の 3 乗であると定義している．これらの情報は，Ninf サーバを経由しリソース情報 DB に登録される．この静的な情報と，実行時に動的に与えられる第 1 引数の値を用いて，スケジューラが実際の計算量，通信料を算出する．

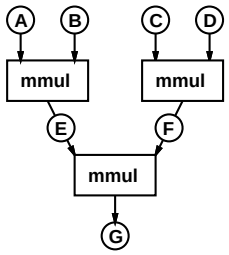

図 5 トランザクションによって生成される Network Task グラフ
Fig. 5 Network Task Graph represented using transaction notation.

4.2 スケジューリング手法

Network Task のグラフをスケジュールするには，計算の配置だけでなく計算間でやりとりされる中間データの配置を行う必要がある．ここでは以下の 2 通りの手法で，計算，中間データの配置を行った．

SimpleScheduling（SIMPLE） 個々の関数を個別にスケジューリングする．その関数を実装しているサーバのうち，最も実行時間が短くなると予測されるものを採用する．データはそのデータを出力する計算サーバに配置する．採用されたサーバはリソース情報 DB に通知され，次のスケジューリングに反映されるため，負荷が特定のサーバに集中してしまうことはない．

FlowScheduling（FLOW） データフローをパスに切り分け，パス単位で配置を行う．基本的に 1 つのパス上に存在する計算/データをすべて同一のプロセッサに配置する．データはそのデータを出力する計算サーバに配置する．上記と同様に，採用したサーバをリソース情報 DB に通知することで，負荷の過度の集中を防ぐ．

FLOW の具体的な手順を以下に示す（**図 6**）．

(1) データフローグラフに含まれるパスの中で最も計算コストが高いものを選ぶ．
(2) 1 のパス中に存在する関数をすべて実装しているサーバのうち，最も「性能の良いサーバ」に配置する．
(3) (2) で配置したパスをデータフローグラフから取り除く．
(4) グラフがなくなるまで 1-3 を繰り返す．

このアルゴリズムは，データ依存関係のある計算をなるべく同一の計算リソースに配置することで，計算サーバ間のデータ通信を削減することを主眼としている．

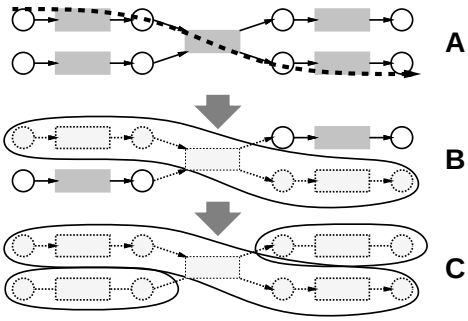

図 6 FlowScheduling の手順
Fig. 6 FlowScheduling.

表 1 dgefa と dgesl の特性
Table 1 Characteristics of dgefa and dgesl.

	通信量（入力）	通信量（出力）	計算量
dgefa	$8n^2$	$8n^2 + 4n$	$\frac{2}{3}n^3$
dgesl	$8n^2 + 12n$	$8n$	$2n^2$

4.3 対象プログラム

実験対象プログラムとして，linpack の dgefa と dgesl のペアを並列して呼び出すプログラムと，行列のカスケード乗算プログラムを用いた．

4.3.1 dgefa，dgesl ペアの並列実行

linpack の dgefa と dgesl はそれぞれ，行列の LU 分解とその結果を用いた求解を行うルーチンである．これらのルーチンを 2 つ連続して呼び出すことで，連立一次方程式の解を求めることができる．両者の通信量/計算量を**表 1** に示す．ここで n は行列のサイズである．これらの式は，Ninf IDL で記述され，Ninf サーバを通してメタサーバシステムに登録されている．スケジューラは，これらの式と，実際のパラメータの値を用いて計算量，通信量を算出して，スケジューリングを行う．

実験では，これらの関数を 1 組として，複数組同時に発行する．このプログラムの Network Task グラフを**図 7** に示す．

表 1 からも分かるとおり，クライアントと dgefa，dgefa と dgesl の間の通信データサイズが，オーダ n^2 と大きい．クライアントと dgefa の間のデータ転送は不可避であるが，dgefa と dgesl の間のデータ転送は同じサーバに配置することで削減することができる．したがって，ペアとなっている dgefa と dgesl を同じサーバに配置することが，スケジューリングのポイントとなる．

4.3.2 行列のカスケード乗算

もう 1 つの対象プログラムとして 2^m 個の行列を m 段でカスケード状に乗算するプログラムを用いた．

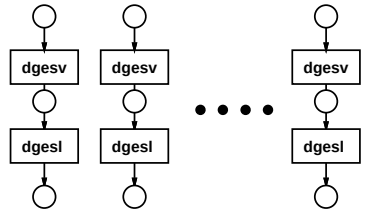

図 7 dgefa，dgesl ペアの Network Task グラフ
Fig. 7 Network Task Graph of dgefa and dgesl pairs.

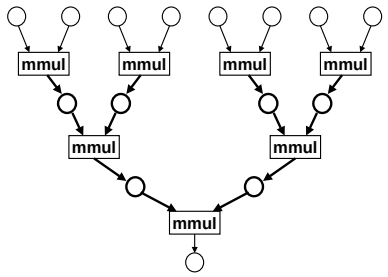

図 8 カスケード乗算の Network Task グラフ
Fig. 8 Network Task Graph of cascaded matrix multiply.

行列サイズを n とすると，計算量は $2n^3$，通信量は入力が $16n^2$，出力が $8n^2$ である．3 段のカスケード乗算を行う際の Network Task グラフを**図 8** に示す．この実験では乗算ルーチンにはサイズに応じた計算負荷という意味しかないのでその質は問題にならないため，ここでは比較的プリミティブなルーチンを用いている．

この計算では，サーバ間の通信量を削減にするためにはすべての計算を 1 つのサーバに配置すればよいのだが，そのような配置では計算負荷が 1 つのサーバに集中し，資源が有効に活用できず性能が低下することが予想される．通信量の削減と負荷の分散をバランス良く取り入れたスケジューリングが必要になる．

4.4 実験環境

実験環境として，つくば市の電子技術総合研究所（ETL），東京都の東京工業大学（TIT），お茶の水女子大（Ocha）の 3 つのサイトを用いた．ETL にクライアントおよびスケジューラなどのメタサーバのモジュールを設置し，TIT にはサーバ A を，Ocha にはサーバ B を設置した．それぞれのサーバの性能，およびサイト間の計測時の通信スループット，ping による通信レイテンシを**図 9** に示す．

メタサーバシステムによるリソースの監視は，スループット/レイテンシの計測は 100 秒おき，サーバのロードアベレージの測定は 200 秒おきに行った．プレディクタによる予測はロードアベレージに関しての

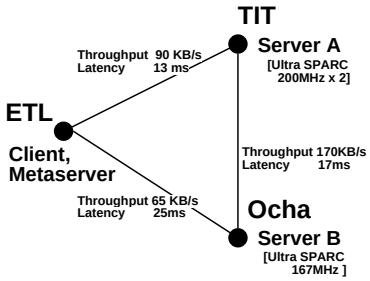

図 9 実験環境
Fig. 9 Experiment environment.

表 2 dgefa，dgesl ペアの測定結果（上段：SIMPLE，下段：FLOW）
Table 2 Mesurement results of dgefa and dgesl (upper: SIMPLE, lower: FLOW).

size	実行時間 [s] 平均/標準偏差	全通信時間 [s] 平均/標準偏差	全計算時間 [s] 平均/標準偏差
100	14.84/4.83	16.80/5.43	0.12/0.00
300	25.75/3.86	52.96/8.99	4.62/0.52
500	64.19/10.51	137.98/22.22	31.36/5.89
700	131.05/14.13	309.65/43.42	90.19/12.50
100	8.74/6.03	5.54/3.72	0.13/0.01
300	16.71/3.79	25.02/6.08	4.58/0.60
500	35.77/6.89	70.73/13.54	29.47/2.29
700	67.98/11.35	119.64/23.70	102.90/8.14

み行った．

4.5 実験結果

4.5.1 dgefa，dgesl ペアの並列実行

測定は，行列サイズを 100，300，500，700，並列して呼び出す dgefa と dgesl のペアを 5 として行った．**表 2** に測定結果を示す．上段が SIMPLE，下段が FLOW の結果である．

実行時間はクライアントが観測した実行開始から終了までの時間である．全通信時間はすべての計算の入出力にかかった時間の総計である．個々の計算を行うサーバが観測した入出力時間を総計した値なので，並列実行時には実行時間全体よりも大きくなることがある．同様に，全計算時間はすべての計算時間の総計である．

全般に，FLOW のほうが実行時間が短縮されていることが分かる．また，実行時間に着目すると，行列のサイズが大きくなるほど，FLOW と SIMPLE の実行時間の差が大きくなっていくことが分かる．これは，FLOW では確実に dgefa と dgesl が同じサーバに配置されるのに対して，SIMPLE では半分以上のケースで別のサーバに配置されるため，サーバ間でデータ転送が生じるが，このスケジューリング失敗に対するペナルティが転送される配列のサイズに従って拡大していくためである．全通信時間の差も同様にサイズに従って大きくなっている．

表 3 カスケード乗算の測定結果（上段：SIMPLE，下段：FLOW）
Table 3 Mesurement results of cascaded multiply (upper: SIMPLE, lower: FLOW).

size	実行時間 [s] 平均/標準偏差	全通信時間 [s] 平均/標準偏差	全計算時間 [s] 平均/標準偏差
100	7.83/0.66	7.84/1.40	4.71/0.12
200	41.13/5.86	35.06/6.74	39.14/3.13
300	116.51/7.12	74.30/10.56	155.14/1.29
100	8.68/2.89	6.91/1.73	4.46/0.47
200	33.11/5.38	24.51/5.75	37.54/4.11
300	99.86/13.65	52.47/10.51	149.85/9.43

表 4 サーバ割当ての比率とサーバ間通信の比率（dgefa，dgesl ペア）
Table 4 Server Invocation ratio and inter-server communication ratio (dgefa and dgesl).

スケジューラ	サーバ A	サーバ B	サーバ間通信
SIMPLE	72.8%	27.2%	53.9%
FLOW	72.8%	27.2%	0%

4.5.2 行列のカスケード乗算

測定は段数を 3 段（8 つの行列の乗算），行列サイズを 100，200，300 として行った（**表 3**）．試行はそれぞれ 10 回行っている．行列サイズ 100 においては，通信量が比較的少ないためか，通信を考慮した FLOW スケジューリングの効果が確認できないが，通信量が大きくなる行列サイズ 200，300 においては，通信時間，実行時間とも短縮されたことが確認できる．

4.6 考察

本節では，両実験におけるサーバの選択の妥当性を検討する．

表 4 に，dgefa，dgesl ペアの並列実行の際に，サーバ A とサーバ B が採用された比率，およびペアとなっている dgefa，dgesl が同一のサーバに配置されなかった比率を示す．

サーバ割当ての比率は SIMPLE も FLOW も完全に同じになった．両者とも CPU 性能とロードに従ってスケジュールしているため，同じ傾向になることは予期したとおりであるが，まったく同一になったのは偶然であると考えられる．なお，サーバ A に割り振られる量がサーバ B の 2 倍以上なのは，図 9 に示すとおりサーバ A は，CPU クロックがサーバ B の CPU よりも高速であるうえ，2 CPU となっているためであると考えられる．また，dgefa と dgesl が同一サーバに配置されない確率は FLOW では 0%であるのに対し，SIMPLE では 53.9%となっており，サーバ間で

表 5 サーバ割当ての比率とサーバ間通信の比率（カスケード乗算）
Table 5 Server invocation ratio and inter-server communication ratio (caccaded multiply).

スケジューラ	サーバ A	サーバ B	サーバ間通信
SIMPLE	57.1%	42.9%	61.7%
FLOW	58.6%	41.4%	20.0%

不要な通信が生じている．これが性能低下の原因であると考えられる．FLOW では不要なサーバ間通信はまったく存在しない．

表 5 に行列のカスケード乗算の際のサーバ A とサーバ B が採用された比率，およびサーバ間通信の生じた比率を示す．行列の 3 段のカスケード乗算においては，サーバ間通信が生じる可能性のある点が 6 カ所存在する（図 8 太線部）．サーバ間通信の比率とは，この 6 カ所のうち何カ所において実際にサーバ間通信が生じたか，すなわち上流の計算と下流の計算が異なるサーバに配置された比率である．このデータは行列サイズ $n = 300$ の場合のものを用いている．SIMPLE では 61.7%存在したサーバ間通信が，FLOW では 21.7%と減少しているのが分かる．これによって，通信時間，ひいては実行時間の削減が実現されている．

以上の FLOW と SIMPLE の比較から，グローバルコンピューティング環境ではデータ転送の削減が非常に重要であり，特に大規模なデータ転送をともなうプログラムの実行においては，データ転送量に着目したスケジューリングアルゴリズムが有効であることが確認できた．また同時に，同一フレームワーク上で複数のスケジューリングアルゴリズムを容易に切り替え，比較検討することが可能であることも確認した．これによって本フレームワークのフレキシビリティが示された．

5. まとめと今後の課題

本稿では，グローバルコンピューティングにおける基本的なスケジューリングフレームワークとして上位スケジューラと下位スケジューラに分離する枠組みを示すとともに，その下位フレームワークの一例として Ninf システムのメタサーバスケジューリングフレームワークの実装を示した．さらにメタサーバシステム上に複数の簡単なスケジューリングアルゴリズムを実装し，本スケジューリングフレームワークの可能性を確認するとともに，スケジューリングアルゴリズムに関する評価を行った．この結果，データ依存関係を考慮しデータ通信量を削減するようスケジューリングを行うことの重要性が明らかになった．また，本フレームワーク上で複数のスケジューリングアルゴリズムが実行可能であることから，本フレームワークが十分にフレキシブルであることが確認できた．

グローバルコンピューティングのためのスケジューリングフレームワークの研究はまだその端緒についたばかりであり，多くの課題が残されている．

フレームワークのシミュレータによる評価 本稿での評価は実環境でのものであるため実験環境の制御が行えず，本フレームワークの本来の目的である，グローバルコンピューティング固有の不安定な環境下での評価を制御された環境の下で行うことができなかった．グローバルコンピューティング環境のシミュレータ[12]などを用いて，再現性のある評価環境の中で，フレームワークの有効性を評価していく必要がある．

高レベルスケジューラの開発 本稿で提唱したスケジューリングフレームワークを活用するには，アプリケーションを Network Task に分割する高レベルのスケジューラが必要になる．

これに関連する研究として，Fortran プログラムからマクロタスクグラフをコンパイル時に構築し，SMP 上にスケジュールする研究がある[13]．この研究が対象としているマクロタスクと本研究の Network Task とでは粒度が異なると思われるが，コンパイル時にグラフを構築する手法としては共通する点があると考えられる．

また，Network Task グラフをユーザが直接記述することも考えられるが，その場合には AVS[14] に見られるような GUI によってユーザをサポートする必要があるだろう．

低レベルスケジューラの整備 本稿で示したスケジューラはサーバ間の通信リソース状態を十分考慮しているとはいえない．より高度なスケジューリングアルゴリズムを実装する必要がある．

上述のマクロタスクグラフの研究は[13]スケジューリングの点でも本研究と関連が深い．グローバルコンピューティング環境と SMP 環境では，通信スループット，レイテンシと計算速度の比率，通信環境，計算環境のヘテロ性，不安定性などの点で大きな相違があるが，これらの SMP 環境下での研究もふまえて，研究をすすめていきたい．

また，今回評価に用いた問題は人工的な問題であり，スケジューリング手法の評価に適切であるとはいいがたい．実用的問題での評価が望まれる．

プレディクタの整備 本稿で用いたプレディクタは，タスクを割り当てたサーバに関して，そのサーバのロードがその後すぐに上昇することを予想する

だけの，アドホックなものである．グローバルコンピューティングの性能評価モデルに基づいたシミュレータ[12]などを用いて，より精度の高い予測を行うことが必要である．

Network Taskの拡張 本稿で提案したNetwork Taskでは比較的単純なデータフローグラフしか表現できず，任意の並列アプリケーションをNetwork Taskに分割することはできない．より一般的なタスク表現形を見い出し，それを扱えるようスケジューリングシステムを拡張する必要がある．

参考文献

1) Ninf: Network Infrastructure for Global Computing. http://ninf.etl.go.jp/.
2) Casanova, H. and Dongarra, J.: NetSolve: A Network Server for Solving Computational Science Problems, *Proc. Super Computing '96* (1996).
3) Grimshaw, A., Wulf, W., French, J., Weaver, A. and Reynolds Jr., P.: Legion: The Next Logincal Step Toward a Natiowide Virtual Computer, *CS*, 94-21, University of Virginia (1994).
4) Foster, I. and Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications* (1997).
5) Berman, F., Wolski, R., Figueira, S., Schopf, J. and Shao, G.: Application-Level Scheduling on Distributed Heterogeneous Networks, *Proc. Super Computing '96* (1996).
6) Weissman, J. and Zhao, X.: Scheduling Parallel Applications in Distributed Networks, *Journal of Cluster Computing* (1997).
7) Czajkowski, K., Foster, I., Kesselman, C., Karonis, N., Martin, S., Smith, W. and Tuecke, S.: A Resource Management Architecture for Metacomputing Systems, *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing* (1998).
8) Feitelson, D. and Rudolph, L.: *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, Vol.1659, Springer (1999).
9) Raman, R., Livny, M. and Solomon, M.: Matchmaking: Distributed Resource Management for High Throughput Computing, *Proc. HPDC-7* (1998).
10) Abramson, D. and Giddy, J.: Scheduling Large Parametric Modeling Experiments on a Distributed Meta-computer, *Proc. PCW '97* (1997).
11) 中田秀基，高木浩光，松岡 聡，長嶋雲兵，佐藤三久，関口智嗣：Ninfによる広域分散並列計算，並列処理シンポジウムJSPP '97論文集，pp.281–288 (1997).
12) Takefusa, A., Matsuoka, S., Nakada, H., Aida, K. and Nagashima, U.: Overview of a Performance Evaluation System for Global Computing Scheduling Algorithms, *Proc. HPDC8*, pp.97–104 (1999).
13) Yoshida, A., Koshizuka, K. and Kasahara, H.: Data-Localization for Fortran Macro-Dataflow Computation Using Partial Static Task Assignment, *Proc. 10th ACM International Conference on Supercomputing*, pp.61–68 (1996).
14) AVS: http://www.avs.com/.

(平成11年9月1日受付)
(平成12年3月2日採録)

中田 秀基（正会員）

昭和42年生．平成2年東京大学工学部精密機械工学科卒業．平成7年同大学大学院工学系研究科情報工学専攻博士課程修了．博士（工学）．同年電子技術総合研究所研究官．現在同所主任研究官．並列プログラミング言語，オブジェクト指向言語，分散計算システムに関する研究に従事．

竹房あつ子（正会員）

昭和48年生．平成8年お茶の水女子大学理学部情報科学科卒業．平成10年同大学大学院理学研究科情報科学専攻修士課程修了．平成12年同大学大学院人間文化研究科複合領域科学専攻博士課程修了．博士（理学）．同年日本学術振興会特別研究員．グローバルコンピューティング，スケジューリング，並列分散処理に興味を持つ．ACM会員．

松岡 聡（正会員）

昭和 38 年生．昭和 61 年東京大学理学部情報科学科卒業，平成元年同大学大学院博士課程中退．同大学情報科学科助手，情報工学専攻講師を経て，平成 8 年より東京工業大学情報理工学研究科数理・計算科学専攻助教授．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インタフェースソフトウェア等の研究に従事．現在進行中の代表的プロジェクトは，世界規模の高性能計算環境を構築する Ninf プロジェクト，計算環境に適合・最適化を目指す Java 言語の開放型 Just-In-Time コンパイラ OpenJIT，制約ベースの TRIP ユーザインタフェース等．並列自己反映型オブジェクト指向言語 ABCL/R2 の研究で 1996 年度情報処理学会論文賞受賞．平成 9 年はオブジェクト指向の国際学会 ECOOP '97 のプログラム委員長を務める．ソフトウェア科学会，ACM，IEEE-CS 各会員．

佐藤 三久（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年，通産省電子技術総合研究所入所．平成 8 年より，新情報処理開発機構つくば研究センターに出向．現在，同機構並列分散システムパフォーマンス研究室室長．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術等の研究に従事．日本応用数理学会会員．

関口 智嗣（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 59 年筑波大学大学院理工学研究科修了．同年電子技術総合研究所入所．情報アーキテクチャ部主任研究官．以来，データ駆動型スーパーコンピュータ SIGMA-1 の開発等の研究に従事．並列数値アルゴリズム，計算機性能評価技術，ネットワークコンピューティングに興味を持つ．市村賞受賞．日本応用数理学会，ソフトウェア科学会，SIAM，IEEE 各会員．