Ninf による広域分散並列計算

中 田 秀 基[†1] 高 木 浩 光[†5] 松 岡 聡[†2]
長 嶋 雲 兵[†4] 佐 藤 三 久[†3] 関 口 智 嗣[†1]

ローカルなネットワーク上でのメッセージパッシングライブラリを用いた分散並列計算はすでに広く行なわれている。しかし、ネットワークの高速化によって現実的になりつつある広域ネットワーク上での分散並列計算については、ソフトウェアの枠組が未だ十分に整備されていない。

我々は、広域分散並列計算に適した分散計算の枠組として「Ninf」を提案している。Ninf は広域分散環境でのマクロデータフローによる並列実行を支援するシステムで、広域での動的負荷分散とスケジューリングを特徴とする。メッセージパッシングライブラリを用いた手法に比較して (1) 広域ネットワークに適した通信パターンを用いる、(2) ユーザにとってプログラミングが容易でかつ再利用性が高い、(3) 既存のライブラリの再利用が容易、(4) ネットワーク上の資源の利用が可能、といった特長をもつ。

Global Parallel Computation using Ninf

Hidemoto NAKADA,[†1] Hiromitsu TAKAGI,[†5]
Satoshi MATSUOKA,[†2] Mitsuhisa SATOH[†3]
and Satoshi SEKIGUCHI [†1]

Distributed computing using message passing libraries in a LAN(Local Area Network) environment is already accepted as an effective supercomputing methodology. On the other hand, although distributed computing in WAN(Wide Area Network) environment is becoming practical due to recent development of high-speed network facilities, software framework for supercomputing in WAN is yet to be established. We propose 'Ninf', a distributed computing framework

for globally distributed computing environment. Ninf enables parallel computing in WAN based on the macro dataflow model, and facilitates automatic dynamic load distribution and scheduling. Ninf has the following advantages over using exisiting message passing libraries in WAN supercomputing: (1) communication protocol suited for globally distributed environment, (2) ease of programming (3) reuse of existing libraries, (4) integration with existing data resources on the Internet.

1. は じ め に

科学技術計算に要求される計算の規模は増大する一方である。これに対して、計算機の単体性能は向上しているとはいえ、使用電力や計算機の構成にかかる金銭的な問題から、単一サイトにおける計算機速度の向上は頭打ちになると考えられる。したがって、要求される計算の規模を満足するだけの計算能力を提供するためには、高速なネットワークを用いて複数のサイトに分散して存在する計算機に計算を分散することが必要になる。現在、LAN 内のワークステーションを用いた分散計算はすでに広く行われているが、広域分散環境では通信速度と計算速度の比率が LAN 内とは著しく異なるため、別の枠組みが必要となる。

我々は、広域分散並列計算に適した分散計算の枠組として「Ninf」を提案している。Ninf は広域分散環境でのマクロデータフローによる並列実行を支援するシステムで、広域での動的負荷分散とスケジューリングを特徴とする。メッセージパッシングライブラリを用いた手法に比較して (1) 広域ネットワークに適した通信パターンを用いる、(2) ユーザにとってプログラミングが容易でかつ再利用性が高い、(3) 既存のライブラリの再利用が容易、(4) ネットワーク上の資源の利用が可能、といった特長をもつ。

本稿では、Ninf におけるユーザアプリケーションのプログラミングモデルについて述べ、メッセージパッシングライブラリを利用する場合に比較して、容易性、再利用性、汎用性、拡張性が高いことを示す。また、広域ネットワーク上での分散計算において効率的な計算を実

†1 電子技術総合研究所 Electrotechnical Laboratory
†2 東京工業大学 Tokyo Institute of Technology
†3 新情報処理開発機構 Real World Computing Partnership
†4 お茶の水女子大学 Ochanomizu University
†5 名古屋工業大学 Nagoya Institute of Technology

現できることを予備的な評価によって示す。

2. 広域分散並列計算 の要件

ワークステーションを LAN で結合したワークステーションクラスタによる分散並列計算は現在広く行なわれている。これらの多くは、 PVM や MPI などのメッセージパッシングパラダイムに基づくシステムを用いている。このパラダイムによる記述には、プログラム間のデータのやりとりの自由度が高いという利点がある反面、データ交換のプロトコル定義やメッセージの明示的な送受信の記述をユーザーが行う必要があるという欠点がある。

広域分散環境下での分散並列処理が、 LAN 上の並列処理と比較して最も大きく異なるのは、単一計算機の計算速度に対するネットワークを介した通信の速度の比率が著しく低いという点である。広域ネットワークの通信速度は向上しつつあるとはいえ、レイテンシに関しては劇的な向上は見込めない。一方で、計算速度は今後も向上し続けることが予想される。このように計算速度が通信速度に比較して非常に高速な場合には、通信のコストが相対的に著しく高くなり、効率的に並列実行できるアプリケーションの型が限定される。短い期間ごとに同期をとりデータを交換する、例えば LU 分解のようなアプリケーションを広域分散環境下で高速化することはむずかしい。計算要素、通信の粒度がともに大きいアプリケーションでなければ効率的な実行は望めない。

科学計算の分野には、このような粗粒度での並列実行に適したアプリケーションが数多く存在する。特定のパラメータをある範囲で走査しながらそれぞれの地点での値を計算するような問題、非常に大量の乱数値に対して特定の演算を行って確率的な解を得るモンテカルロシミュレーションのような問題などである。これらの問題は、相互通信を必要としない多数の独立した計算で構成されており、基本的にオーバヘッドなく任意のサイズに分割し並列に実行することができる。

このような問題を粗粒度並列で記述するには、メッセージパッシングのようなパラダイムは不適である。メッセージパッシングは、データ交換の頻度がある程度高く、複雑な通信形態が必要とされる場合を前提にしている。このため、非常に簡単な通信形態を記述する場合にも、プロトコルの設計や明示的なメッセージ送受信をユーザが行う必要がある。これは、ユーザにとって大きな負担となる。簡単な通信形態を、簡単に記述できる枠組みが必要である。

従来のメッセージパッシングシステムのもう一つの問題点として、構成要素計算機と通信系の信頼性の問題がある。従来のメッセージパッシングシステムは、すべての計算機が同一 LAN 内に存在し、それらすべてがある程度信頼できる形で運用されていることを前提としている。しかし広域分散環境では、通信回線の信頼性は低く、各々の計算機の管理体制も異なることを前提としなければばならない。広域分散環境でのシステムには、このような環境でも頑健に運用できることが要求される。PVM や MPI などのメッセージパッシングライブラリでは、通信相手となるホストを適切に広域環境上で指定する方法は定義されておらず、かつ対故障性なども考慮されていないので、これらの要求は満たされていない

3. Ninf による広域分散並列計算

3.1 Ninf システムの概要

我々は、前節で述べたような要求を満たすシステムとして、 Ninf システムを設計した[1)]。 Ninf は計算を粗粒度な要素ルーチンに分割してネットワーク上に分散した計算機に配置し、それらの間にデータフローを構成し、計算を行なうシステムである。

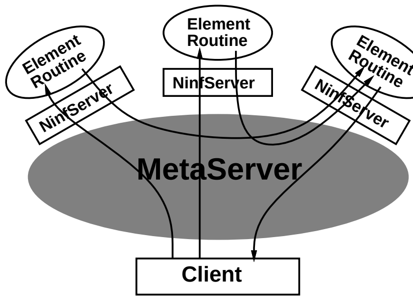

図1 Ninf システム

Ninf システムは、クライアント、 Ninf サーバ、要素ルーチン、メタサーバの 4 つの要素からなる。要素ルーチンは独立したプログラム部品で、再利用可能な一般的なインターフェイスを与えて設計する。要素ルーチンは複数の入力を得、複数の出力を与えるだけのもので、状態は持たない。状態を持たないため、要素ルーチンのインターフェイスは非常に単純になる。この要素ルーチンを組み立てる枠組みとして、クライアントプログラムを記述する。クライアントプログラムには、使用する要素ルーチンとその間のデータ依存関係を記述する。データ依存関係が最も単純な場合、すなわちクライアントと一つの要素ルーチンの間にしか依存関係がない場合が通常のクライアント / サーバモデルになる。

Ninf サーバは分散システムを構成する各々の計算機上で待機し、それぞれの計算機上の要素ルーチンを管理する。リクエストに応じて個々の要素ルーチンを実行する役割を持つ。メタサーバは、クライアントと Ninf サーバを繋ぐ役割を果たす[2)]。クライアントプログラムに記述された要素ルーチン間のデータ依存関係は、クライアントの Ninf ライブラリによってメタサーバに転送される。メタサーバはデータ依存関係に基づいて、スケジューリング、負荷分散を行ない、 Ninf サーバへリク

エストを発行し要素ルーチンを実行する。要素ルーチン間のデータ転送もメタサーバの役割の一つである。

各計算要素ルーチンの記述は、通常の C または Fortran の関数の記述と同様に行なうことができる。既存のライブラリに簡単なインターフェイス記述を加えるだけでそのまま使用することができるので、各専門領域における膨大なソフトウェア資産を活かすことができる。各要素ルーチン間には、要素ルーチン同士がメモリ空間を共有しているかのように見せるプログラミングインターフェイスが提供される。データフローの上流の計算要素が計算結果を自分のメモリに書き込むと、その値が下流の計算要素のメモリに自動的に転送される。このため、メッセージパッシングのように、プログラマが明示的に通信を記述する必要はなく、各計算要素は通常の単一ノードでの逐次プログラムと全く同じように記述することができる。このような簡潔なインターフェイスを採用したことで、プログラマの負担が軽減されている。通信プロトコルの設計ミスによるデッドロックなどのバグが発生する可能性はない。

Ninf では問題の記述を、要素ルーチン自身の記述とデータ依存関係の記述という 2 つのレベルに明示的に分離する。このため、システムとしての汎用性、拡張性に富んでいる。例えば、要素ルーチンの各入力のソースは、プロトコルを満たしてさえいれば何でもかまわない。たとえば、 WWW や ftp から取得したデータを、計算の入力として直接用いることも可能である。

3.2 インターフェイス の記述

要素ルーチンのインターフェイスは Ninf IDL(Interface Description Language) と呼ばれる言語で記述する。この記述をインターフェイスジェネレータでコンパイルして、 Ninf の提供するライブラリとのインターフェイスの役割を果たすスタブルーチンを作成する。このスタブと関数本体をリンクして、要素ルーチンの実行形式を得る。 Ninf サーバはリクエストに応じてこの実行形式を実行する。

インターフェイスでは、各引数の入出力モードや配列の長さを指定する。例として、行列の加算を行うルーチンのインターフェイスを示す。

```
Define madd(mode_in int n,
  mode_in double A[n][n],
  mode_in double B[n][n],
  mode_out double C[n][n])
```

インターフェイス宣言のシンタックスは、それぞれの引数に型の宣言の他にモードの宣言が必要である点を除けば、 C の関数宣言に準ずる。また、返り値という概念はない。この例では第 1 引数が行列のサイズを示している。第 2、 3 引数が要素ルーチンへの入力、第 4 ルーチンが出力となっている。

ここで注意して欲しいのは、配列の長さを第一引数で指定している点である。このように動的に変化する転送長を指定できるのは Ninf の特徴の一つである。さらに、配列のサイズとして式を記述することもできる。例えば、上記の第 2 引数を

```
mode_in double A[n * n],
```

と記述することもできる。さらに配列の転送範囲を、開始点、終了点、ストライドで指定することもできる。これらの機能によって、柔軟なインターフェイスの記述が可能となっている。

3.3 Ninf のプログラムインターフェイス

Ninf クライアントは現在、 C, Java, Lisp 環境に実装されているが、ここでは C 言語の環境を説明する。

Ninf システムを用いて計算をリモートの計算機で実行するには、以下のよう書く。

```
Ninf_call(func_name, .....);
```

`Ninf_call` の第一引数には URL に準じた記法でサーバのホスト名 / ポート、計算を指定する。

`ninf://`*hostname*[*:port*]`/`*funcname*

サーバ、ポート名を省略し、関数名だけを記述した場合にはシステムのデフォルトとして指定されたメタサーバが、適切なホストを選択する。

第 2 引数以降には通常の C の関数呼び出しと同様に引数を記述する。

この Ninf 呼び出しはブロッキングである。すなわち、リモート計算機での計算が完全に終了し、計算結果を回収するまでリターンしない。これでは、複数の Ninf 呼び出しを同時に発行することができないので、計算の発行と計算結果の回収を分割して行なう方法が用意されている。

```
Ninf_call_async(func_name, ...);
```

この関数は、要素ルーチンの実行と引数の送信を行ない、計算の終了を待たずにリターンする。返り値として、その Ninf 呼び出しのセッションの ID を返す。

計算の結果を回収するには、この ID を引数として

```
Ninf_wait(ID);
```

を呼び出す。このルーチンはそのセッションの計算が完了するのを待ち、すべての結果を回収し`Ninf_call_async` の引数に書き込んでリターンする。これらを用いると、

```
id1 = Ninf_call_async(func_name, ...);
id2 = Ninf_call_async(func_name, ...);
    :
Ninf_wait(id1);
Ninf_wait(id2);
    :
```

のように記述して、複数の計算を並行して発行することができる。

すべてのセッションの終了を一つの関数で待つこともできる。

```
Ninf_wait_all();
```

とすると、すべてのセッションのが終了するのを待ち、解を回収してくれる。ここで重要なのは、Ninf_wait_all は計算が終了したセッションから順番に

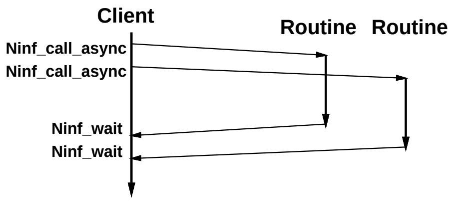

図2　Ninf_wait

解の回収を行うことである。このため、各セッションの実行終了時間が予測できないような場合にも効率よくバリア待ちを行うことができる。

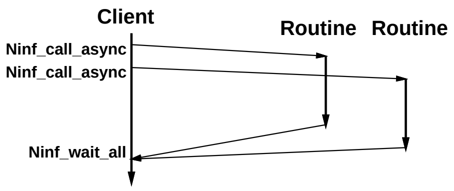

図3　Ninf_wait_all

3.3.1　トランザクション

`Ninf_call_async` を用いれば互いに依存関係のない単純な並行計算は容易に記述できる。しかし Ninf 呼び出しの間に依存関係がある場合には、適切な場所に同期を挿入することが必要となる。これは並列プログラミングに不馴れなプログラマにとっては容易ではない。そこで、データ依存関係にしたがった発行と同期を自動化するためにいくつかの連続する Ninf 計算呼びだしをまとめた、トランザクションという機構を導入した。トランザクションは開始点と終了点を指定することで、ユーザによって明示的に設定される。トランザクション内の複数の`Ninf_call` は依存関係が自動的に解析され、その依存関係が保証されるようにスケジュールされ、並列実行される。

例えば、行列 A、 B、 C、 D の加算は以下のように単純に記述できる。

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

実際の計算の発行は、トランザクション終了時に一括して行なわれる。トランザクションの内部では、`Ninf_call` のセマンティクスが変化し、実際の Ninf 呼び出しを行わず、データ依存グラフの構築のみを行い、すぐにリターンするようになる。

トランザクションが終了すると、クライアントのライブラリが、データ依存グラフと各 Ninf 呼び出しにおける引数とをメタサーバに転送する。メタサーバは、データ依存グラフに従って、各 Ninf 呼び出しをスケジューリングし、 Ninf サーバへのリクエストを行なう。この例の場合 1 番め、 2 番めの Ninf 呼び出しが終了したのち、 3 番めの Ninf 呼び出しが発行される。

C 言語環境でのデータ依存解析は、配列データのポインタの同一性と Ninf 呼び出しの各引数の入出力モードに基づいて行なう。前述したように、 Ninf の各要素ルーチンは自分のインターフェイス情報を持っており、 Ninf 呼び出し時にこの情報が動的に転送される。このインターフェイス情報にモード情報も含まれており、データ依存解析にはこの情報を用いる。例の場合、madd の第 3、 4、 5 引数の入出力モードはそれぞれ、入力、入力、出力であり、図 4 のようなデータ依存関係があることがわかる。

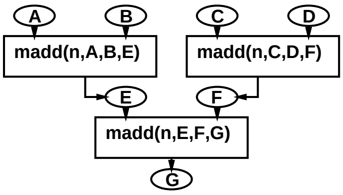

図4　行列加算のデータ依存関係

このとき中間データのクライアントへの転送は抑制される。上記の例でいえば、変数`E,F` が中間データである。これらの値はデータ依存関係を表現するために用いられたに過ぎず、クライアントプログラムで必要とされる値ではないためクライアントに書き戻さない。中間データは実行時に自動的に検出される。

3.4　コールバック

Ninf には 要素プログラムから、クライアント側のルーチンを呼び出す機能、コールバックが用意されている。この機能は、例えば要素プログラムが連続的に計算を行い、中間結果をクライアント側で表示したい場合や、計算ルーチンの一部を動的に切り替えたい場合などに用いる。コールバックを用いると要素ルーチンのコンテクストを保持したままでの、クライアントとインタラクションが可能になる。

コールバックを用いるには、 IDL にコールバック関数のインターフェイスを定義する必要がある。定義のシンタックスは 要素ルーチンの定義に準ずるが、要素ルーチンの引数として記述する。

例として要素ルーチン calc から、クライアント側のルーチン display_real を呼び出して画面を表示させることを考える。 IDL を以下のように記述する。

```
Define calc(mode_in ....,
            display(mode_in ...));
```

この場合、 display がコールバックルーチンで要素ルーチンの内部では、この名前でルーチンを呼び出す。

呼び出し側のコードは下のようになる。

```
Ninf_call(...., display_real);
```

`display_real` はコールバックルーチンの関数ポインタである。この実行の様子を図 5に示す。

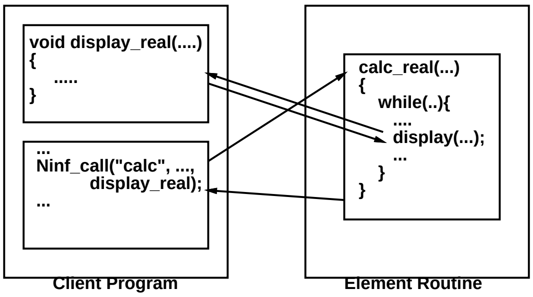

図5　コールバック

一つの要素ルーチンから呼び出せるコールバックルーチンの数には制限はない。用途に応じて複数のコールバックルーチンを使い分けることもできる。

また、非同期呼び出しを用いて複数のセッションを同時に実行している場合にも、コールバックを用いることができる。`Ninf_wait_all` は select システムコールを用いて実装されており、すべてのセッションからのコールバックを同時に待ち受けることができる。この機能を用いると後に示すように動的な負荷分散を行うことが可能である。

3.5　メッセージパッシングライブラリとの比較

Ninf による記述と代表的なメッセージパッシングライブラリの一つである PVM による記述とを比較してみよう。ここでは、NAS Parallel Benchmark の一つで、モンテカルロ法によって円周率を求める計算である EP を記述してみる。

図 6に Ninf による EP のコア部分を示す。

```
for (i = 0; i < PU; i++){
  Ninf_call_async(buffer, i, NPP, ktmp+i,
    xsumtmp+i, ysumtmp+i, qtmp[i]);
}
Ninf_wait_all();
```

図6　Ninf による EP の実行

ループの中で複数の Ninf 呼び出しを行い、ループの後で一括してセッションの終了を待っている。Ninf では、このように簡潔なプログラムを記述するだけで、ep のサブプログラムが自動的に適切な要素計算機上に分散され、並列に実行される。

これに対して、PVM ではプログラマが、プロセスをフォークするコード、解を回収するコードをいちいち手で記述しなければならない。図 7に PVM でのスレーブプロセスのフォークのコードをしめす。

フォークの際にはホストネームを明示的に与える必要がある。さらに、ここでは省略するが解の回収のコードも必要となり、本来プログラムの本質的でない部分である通信部分の記述がプログラマにとって大きな負荷になることがわかる。

もう一つの問題点は、再利用性である。メッセージパッシングライブラリでは、プログラムの用いているプロトコルはプログラムに非明示的に埋め込まれてしまうため、要素プログラム自体に一般性があったとしてもそれを再利用することが難しくなる。Ninf では、要素プログラムのインターフェイスは IDL ファイルとして独立して記述されるため、要素プログラムの再利用は非常に容易である。

```
for(i=0; i<PU-1; i++){
  if(pvm_spawn(PROG,NULL,TASK_CREATE,
     hname[(i%hnum)],1,&tids[i+1])!=1){
    perror("Cannot create process\n");
    pvm_exit();
  }
}
```

図7　PVM でのフォーク

4. 評　　価

Ninf による分散並列実行の予備評価として、LAN 内での並列実行を評価した。

4.1　実 験 環 境

実験環境としては、DEC Alpha (333 MHz) の 32 台のワークステーションクラスタを用いた。各ワークステーションは、100Base/T のスイッチで結合されている。32 台の他にクラスタのホストとして 1 台の Alpha ワークステーションがある。これらは NFS でファイルを共有している。

Ninf サーバをそれぞれのワークステーションで実行し、クライアントには、ホストのワークステーションを用いた。今回はトランザクション を用いず、非同期の Ninf 呼び出しを用いて並列実行した。

4.2　EP による評価

前述した EP を用いて評価を行った。この計算では、乱数による試行は完全に独立に行なうことができ、さらにそれぞれの試行一回あたりのコストは一定であるから、容易に負荷を等分することができる性質のよいプログラムである。

実験は、Sample と ClassA と ClassB とで行った。Sample では 2^{24} 回、ClassA では 2^{28} 回、ClassB では 2^{30} 回の試行を行う。問題の分割法としては、単純に試行回数をプロセッサ数で割って各プロセッサに割り付けた。割り付けは明示的にプロセッサを指定して行った。32 プロセッサの場合各要素ルーチンでの指向回数は Sample、ClassA、ClassB でそれぞれ、2^{19} 回、2^{23} 回、2^{25} 回の実行が行われることになる。図 8に実行結果を示す。

Sample では、32 台の時に大きく性能が低下している。ClassA、ClassB ではほぼリニアに速度が向上しているが、ClassA では 32 台でわずかに低下が見られる。これは問題が細分化され、個々の要素ルーチンでの

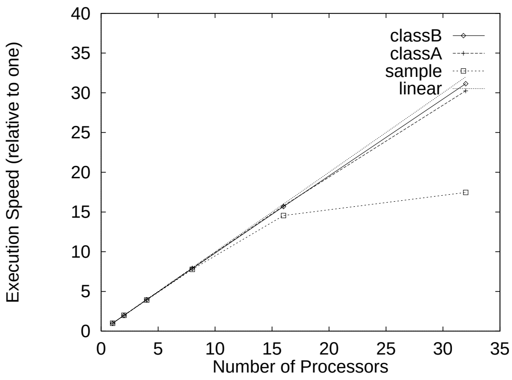

図8　EP の実行

実行時間が短くなった結果、Ninf によるオーバーヘッドが顕在化したものであると思われる。

分散台数を常に 32 台にし、個々の要素ルーチンでの試行回数を 2^{15} から 2^{25} まで変化させた結果を図 9に示す。値はローカルに同じ試行回数を実行した場合との比較である。

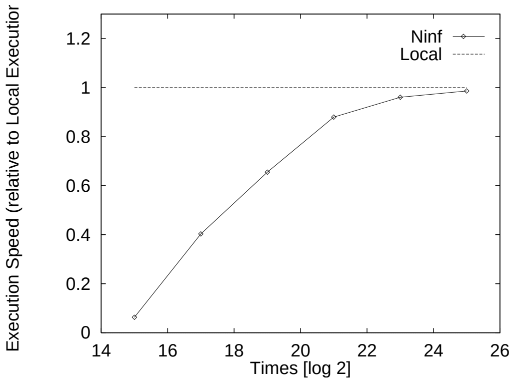

図9　試行回数と相対実行速度

Ninf の呼び出しによるオーバーヘッドは分散台数にのみ依存し、この場合は固定であると考えて良い。試行回数が増え要素ルーチンの実行時間が延びるにしたがって、Ninf 呼び出しのオーバヘッドが無視できるようになるのがわかる。

4.3　メタサーバによる負荷分散

表 1にサーバを明示的に指定して負荷分散をおこなった場合と、メタサーバを用いて EP の負荷分散を行った場合との比較を示す。

台数が 16 台までなのは、現在使用しているカーネルの、プロセス当りのファイルディスクリプタの数の制限のためである。メタサーバでは台数の 2 倍のコネクションを使用するため、ファイルディスクリプタを大量に消費してしまう。このため、32 台に対してはスケジューリングが実行できなかったためである。

メタサーバによるスケジュールのコストは使用台数に

表1　実行時間 (秒)

	1	2	4	8	16
Sample					
メタサーバ	35.7	18.1	9.4	6.2	6.9
サーバ指定	35.3	17.7	8.8	4.4	3.3
ClassA					
メタサーバ	565.6	283.4	141.6	71.1	39.2
サーバ指定	565.4	282.4	141.2	70.6	37.2
ClassB					
メタサーバ	2261.0	1132.1	566.3	283.3	141.7
サーバ指定	2259.0	1129.5	564.8	282.4	141.6

比例して増大している。また、問題サイズには依存しないため問題サイズが大きくなるにつれて相対的に無視できるようになっている。

4.4　状態密度計算

この計算は巨大分子の状態密度を、行列の対角を行うことなく求める手法である。この状態密度計算 (density of states)[3] は、巨大行列をバネのつながった力学モデルに見立て、周期的な外力をあたえ、バネの共鳴の度合いを状態数とするものである。この問題は外力の周波数に対して並列化を行うことができるため、並列計算に適してはいるが、各周波数における負荷が一定でないため、均等に負荷を分散することは困難である。図 10に負荷の分布を示す。

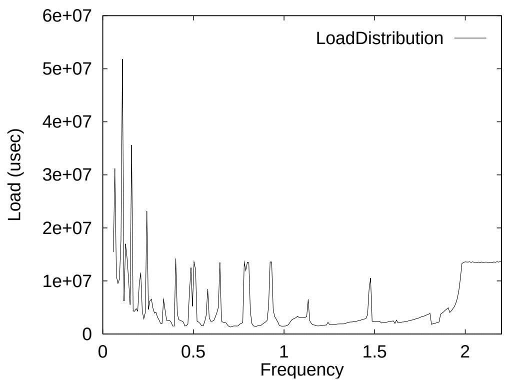

図10　状態密度計算の負荷分布

横軸が周波数、縦軸がその周波数の共鳴度合いを計算するための計算時間である。負荷が周波数によって著しく異なるため、周波数に対して単純にブロックやサイクリックに分割すると、負荷が均等に分割されないため、台数効果が得られない。さらに、各周波数における負荷は事前に予測することは不可能である。

そこでコールバックを用いた動的な負荷分散をおこなった。クライアント側のルーチンでは、サーバ側の要素ルーチンを演算すべき周波数を指定せずに立ち上げる。さらに、演算すべき周波数を供給するコールバックルーチンを用意する。サーバ側の要素ルーチンは、コールバックで演算を実行すべき周波数を取得して、その周

表2　実行時間 (秒)

実行時間 [秒]	1	2	4	8	16	32
サイクリック	1380.4	731.7	387.1	233.8	165.3	94.4
動的負荷分散	1385.9	697.6	351.8	179.8	91.3	52.9

波数での演算結果を次のコールバックでクライアントに返す。これを周波数が取得できなくなるまで繰り返す。

図 11にコールバックによる実行の台数効果を示す。比較のため、単純なサイクリック分割を用いた実行も行った。

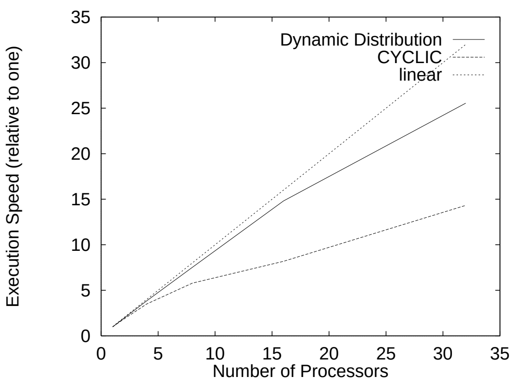

図11　状態密度計算の台数効果

サイクリック分割の場合は、負荷の不均衡によって台数に応じた性能が出ていない。これに対して動的負荷分散の場合には 32 台で 25.5 倍の性能が得られている。完全にリニアなスピードアップが得られない理由は、完全に負荷を均衡させせることは本質的に不可能であるため、負荷を分散するコールバックルーチンがボトルネックになっているため、の 2 点であると考えられる。

表 2に実行時間を示す。 1 プロセッサの場合の実行時間は、サイクリックの方が 5.5 秒程度早くなっている。これはコールバックのコストである。この場合、周波数帯域を 256 分割しているのため 256 回のコールバックが生じている。

図 12に 32 台で実行した場合のブロック、サイクリック、動的分割の各場合におけるそれぞれのプロセッサに割り付けられる負荷量を示す。それぞれの場合の実行時間はそれぞれのグラフの最も高い値になる。動的分割では負荷が比較的均等に分配されており、実行時間が短縮されるのが分る。

4.5　PVM との比較

図 13に PVM で記述した場合に対する、 Ninf で記述した場合の実行速度の比較を示す。対象プログラムは前出の EP である。

PVM によるプログラムは、問題の分割手法、通信のプロトコルが Ninf によるものと同じになるよう記述した。 Ninf で記述したものは、 PVM で記述したものと

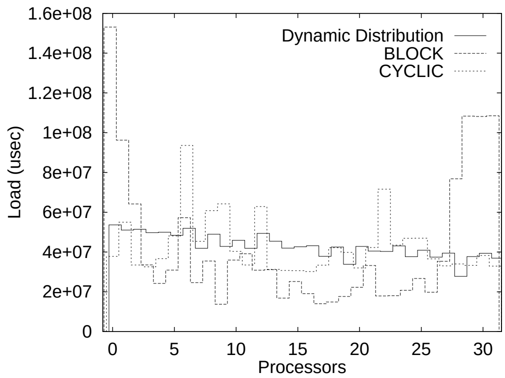

図12　コールバックによる負荷分散

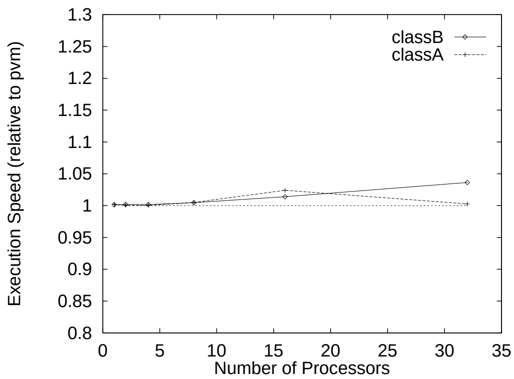

図13　PVM を用いた 場合との比較

ほぼ等価か、より少ないオーバヘッドで実行できていることが分る。

4.6　広域環境下での性能低下予測

今回の評価では、クライアントのマシンと Ninf サーバのマシンとは高速な LAN で接続されている。本節では、広域環境で通信速度が低下し Ninf によるオーバヘッドが大きくなった場合の実行速度について考察する。

通信速度の低下によりオーバヘッドは増大する。しかし、問題の計算時間が充分長いため実行速度の面ではほとんど問題にならない。先ほどの ep を例にとる。ClassB を 32 台で演算した場合の試行回数は 2^{25} である。これを実行するには、およそ、 67.3 秒ほどかかる。この場合の Ninf のオーバヘッドは 1 秒弱に過ぎない。したがって、このオーバヘッドが仮に 10 倍になったとしても、オーバヘッドは全実行時間の 1/8 程度に過ぎない。

オーバヘッドが 2 倍、 5 倍、 10 倍、 20 倍であったとしたときの、ローカルな場合との相対的実行速度を図 14に示す。

ここで示したアプリケーションは、 Ninf のターゲットとしては比較的要素ルーチンの実行時間の短いアプリケーションである。しかし、このような場合でさえオー

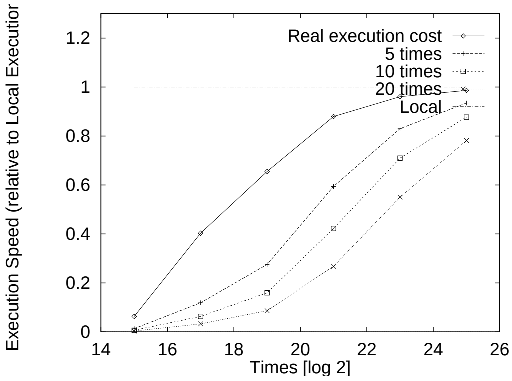

図14　オーバヘッドが大きい場合の相対実行速度

バヘッドが増大したとしても計算時間のほうが支配的である。アプリケーションの規模が増大すれは、さらに計算時間が支配的になるので、 Ninf のオーバヘッドは広域環境下でも問題にならないと思われる。

5. 関連研究

分散環境でのスーパーコンピューティングを行うプロジェクトは、幾つか行われつつある。 NetSolve[4] は、Ninf によく似たシステムで、クライアントの API も類似している。 NetSolve にはデータフローを解析し、スケジュールする機能がない点が Ninf と異なる。 API のセマンティクスはほぼ等価なので、 NetSolve で用意された数値演算ライブラリを Ninf から使用し、 Ninf で用意した数値演算ライブラリを NetSolve から使用する、「相互乗り入れ」を検討中である。

スイス ETH の RCS(Remote Computation System)[5] は、 Ninf と同様にバックエンドのスーパーコンピュータをリモートから使用するシステムである。Ninf との相違としては、 RCS では下位のレイヤとして PVM を用いており広域分散に適していないこと、動的なサーバの追加ができないこと、並列計算に対応していないことが挙げられる。また、 Ninf ではすべてのリモートルーチンを一つの API で呼び出すことができ、ルーチンの追加時にクライアントの変更が必要ないのに対して、 RCS ではリモートルーチンを追加するごとに別の関数をインターフェイスとして追加する。このため、システムの拡張性にとぼしい。

早稲田大学のメタコンピュータシステム[6] は、自動並列化 FORTRAN コンパイラを用いて問題を分割し、リモートホストに予め稼働させておいた Resource Management Server にプログラムをソースコードで転送、サーバ上で実行形式を作成し、実行するシステムである。コードを転送することによって生じるセキュリティ問題をどう回避するかがが技術的な問題点である。

Mentat[7]、 Legion[8] は C++ の拡張言語を用いる並列分散言語である。拡張言語で記述されたプログラムは、オブジェクトごとに分散され並列に実行される。Ninf は言語を新たに設計することを避け、クライアントを単なるライブラリとして提供することで、ユーザの負担を減らすことを重視しており、これらのシステムとことなる。

6. おわりに

広域分散並列計算に適した分散計算の枠組への要請について議論し、我々のシステム Ninf がその要請を満たすことを示した。また、ワークステーションクラスタを用いた予備的な評価を行い、 Ninf が分散環境において効率的な計算を実現でき、 PVM に遜色ない実行性能であることを示した。

今後の課題としては、広域分散に必要とされるロバストネスの向上が挙げられる。また、今回用いたメタサーバのスケジューリング手法は単純なラウンドロビンである。これを洗練することも今後の課題である。

参考文献

1) Sekiguchi, S., Sato, M., Nakada, H. and Nagashima, U.: – Ninf – : Network base information library for globally high performance computing., *Proceedings of Parallel Object-Oriented Methods and Applications (POOMA)* (1996).
2) 中田秀基, 草野貴之, 松岡聡, 佐藤三久, 関口智嗣: ネットワーク数値ライブラリ Ninf におけるメタサーバアーキテクチャ, 情報処理学会研究報告 HPC, Vol. 96, No. 22 (1996).
3) 日向寺祥子, 長島雲兵, 細矢治夫, 関口智嗣, 佐藤三久: 大規模実対称行列の状態密度の計算とその並列化, 情報処理学会研究報告 HPC, Vol. 93, No. 93-HPC-48, pp. 57–64 (1993).
4) Casanova, H. and Dongara, J.: NetSolve: A Network Server for Solving Computational Science Problems, *Proceedings of Super Computing '96* (1996).
5) Arbenz, P., Gander, W. and Oettli, M.: The Remote Computation System, Technical Report 245, ETH (1996).
6) Shudoh, K. and Muraoka, Y.: Parallelizing and Distributing Compilers for Meta-Computer Systems (1996). http://www.muraoka.info.waseda.ac.jp/sc96/.
7) Grimshaw, A.S.: Easy to Use Object-Oriented Parallel Programming with Mentat, *IEEE Computer*, pp. 39–51 (1993).
8) Grimshaw, A. S., Wulf, W. A., French, J. C. and Alfred C. Weaver, P. F. R. J.: Legion: The Next Logincal Step Toward a Natiowide Virtual Computer, CS94-21, University of Virginia (1994).