ハイパフォーマンスコンピューティング 60－14
（1996. 3. 4）

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

中田秀基[†] 草野貴之[††] 松岡 聡[††]
佐藤三久[†] 関口智嗣[†]

ネットワーク数値情報ライブラリ Ninf (Network based Infomation library for High Performance Computing) は、高速なネットワークを前提として、主に数値演算の分野において、計算自体を多くのユーザに提供することを目指したシステムである。

本稿では、Ninf システム を構築する要素の一つである メタサーバに関して、そのアーキテクチャを示し、簡単な性能予備評価を示す。メタサーバは、サーバとクライアントの間にたちサーバの場所をクライアントに対して隠蔽する役割を果たす。また、メタサーバを用いることにより、簡単な分散並列計算が可能になる。

A Meta Server Architecture for Ninf: Networked Information Library for High Performance Computing

HIDEMOTO NAKADA,[†] TAKAYUKI HUSANO,[††]
SATOSHI MATSUOKA,[††] MITSUHISA SATOH[†]
and SATOSHI SEKIGUCHI [†]

To establish a framework of information sharing in the numerical computation area, we have proposed the Ninf,Network based information library for high performance computing.

In this paper, we show a Meta Server architecture, which is a component of the Ninf system. Meta Server stand between the Server and the Client and hides the Server from the Client. It also enables easy distributed concurrent computation.

1. はじめに

コンピュータネットワークの発展にともない、ネットワークを利用した情報提供サービスが盛んに行なわれている。これらのサービスの特徴は、情報の実際に存在する位置の透明性であり、情報資源の仮想的共有化であると考えることができる。計算科学技術、数値解析技術領域における情報資源の共有化の手法として、このような情報資源の共有のための枠組として、われわれは Ninf (Network based Infomation library for High Performance Computing) を提唱してきた[1)]。

Ninf は基本的にはサーバ・クライアントモデルのシステムである。ネットワーク上に数値情報や計算自身を提供するサーバを設け、ユーザにはサーバに接続するクライアント機能を持つライブラリを提供する。ユーザはこのライブラリを自分のプログラムにリンクすることで、ほとんど、ネットワークの存在を意識することなく、数値情報や計算の提供を受けることができる。

単純なサーバクライアントモデルでは、クライアントは何らかの方法でサーバを知る必要がある。しかし、Ninf のサーバの数が増加すると、サーバのアドレスやそれぞれが提供する計算をユーザが管理することは容易ではない。

このため、Ninf システムの一部を担うものとして、メタサーバと呼ぶものを導入した。メタサーバは、Ninf のサーバとクライアントの中間に存在し、ユーザに対して実際に計算が行なわれる場所を隠蔽する役割をになう。さらに、分散環境での並列計算を実現する機能を持つ。

本稿では、このメタサーバのアーキテクチャについて述べ、予備的な性能評価を行なう。

2章で Ninf システムを概説し、3章で Ninf Meta Server の設計方針を述べ、4章で実装について述べる。5章で性能の予備評価を行ない、6章で考察を行なう。

† 電子技術総合研究所
Electrotechnical Laboratory
†† 東京大学工学部
Faculty of Engineering, The University of Tokyo

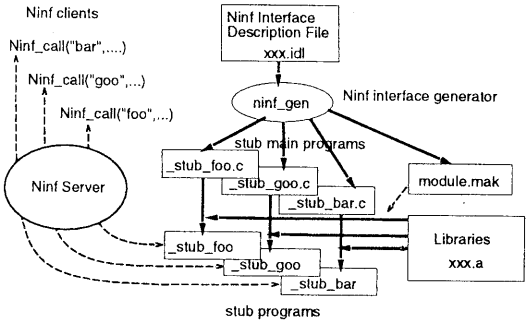

図1 インターフェイス ジェネレータ
Fig. 1 Interface Generator.

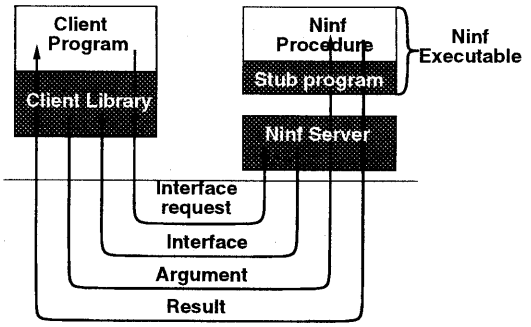

図2 Ninf RPC
Fig. 2 Ninf RPC.

2. Ninfシステムの概要

2.1 Ninf サーバと Client

Ninf はサーバ・クライアントモデルのシステムである。ネットワーク上に数値情報や計算自身を提供するサーバを設け、ユーザにはサーバに接続するクライアント機能を持つライブラリを提供する。サーバ、クライアント間は、独自に設計開発した Ninf RPC と呼ぶ RPC プロトコルを用いて通信する[2)]。計算などを提供するサーバを Ninf サーバと呼ぶ。

Ninf RPC は、通常の C 言語で技術計算を行なうユーザーが、容易に移行できることを目的に設計され、以下の特徴を持つ。

- 計算のインターフェイス情報を動的に供給する。
- メモリ空間を共有するイメージを提供する。共有するサイズを動的に決定することができる。
- 数値演算に対象を絞った単純な構造をもつ。

例として、行列の乗算を計算するルーチンを考えてみよう。典型的な呼びだしのインターフェイスは以下のようになるであろう。

```
double A[N][N],B[N][N],C[N][N];
mmul(N,A,B,C);
```

Ninf ではこの関数を以下のような形で呼び出すことができる。

```
Ninf_call("mmul",N,A,B,E);
```

このように、通常の C 言語での呼び出しとほとんど同じ形式で Ninf の関数を呼び出すことが可能である。

計算ライブラリの提供者は、計算ルーチン自身とそのインターフェイス情報を提供する必要がある。インターフェイス情報は Ninf IDL と呼ぶ IDL(Interface Description Language) で記述する。Ninf IDL ではデータの型を基本型とその配列のみに限定しているため簡潔であり、インターフェイス情報の記述は容易である。IDL 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、インターフェイス情報を埋め込んだスタブプログラムを得る。このスタブと計算ルーチンをリンクしたものが実行ファイルとなる (図 1)。

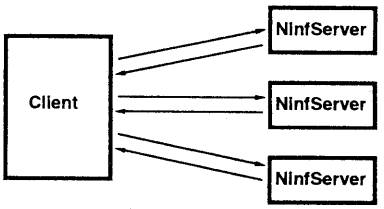

図3 Ninf サーバのみによる構成
Fig. 3 Configuration without Meta server.

Ninf RPC による実行は以下のように行なわれる。

(1) クライアントが Ninf サーバに計算のインターフェイス情報をリクエストする。
(2) Ninf サーバがインターフェイス情報をクライアントに渡す
(3) インターフェイス情報に基づき、クライアントは転送するべきメモリ領域を決定し転送する。
(4) Ninf サーバは計算ルーチンをスタブとリンクした実行ファイルを exec し、転送されてきたデータを exec したプロセスにフォワードする。
(5) 計算の結果を逆方向に転送する。

他の RPC との最大の差異は、インターフェイス情報を事前にクライアント側とサーバー側で共有している必要がないことである。このため、ユーザが RPC を使用するためには、対象の名前を知っているだけでよく、ユーザの負担が小さい。

図 2に Ninf RPC の実行の様子を示す

2.2 メタサーバの必要性

Ninf サーバとクライアントがあれば、Ninf の目的とする分散環境における計算自信の供給は基本的には可能である。しかし、単純なサーバ、クライアントシステムでは以下のような問題が生じる。

- ユーザが、Ninf サーバのアドレスを知らなければならない。
  Ninf サーバは、提供する計算を実行する計算機に存在しなければならない。Ninf では複数の計算機がそれぞれ得意な計算を提供することを前提としているので、Ninf サーバの数は増加することが予想される。増加した Ninf サーバのアドレスをユーザ

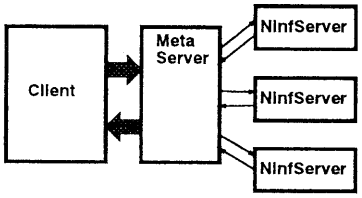

図4 メタサーバを用いた構成
Fig. 4 Configuration with Meta Server.

が把握するのはむずかしい。また、ユーザーが一つのプログラムで複数の計算を行なう際に、一つのNinf サーバがすべての計算をサポートしていなければ、それぞれの計算にあわせて Ninf サーバを切替えなければならなくなる (図 3)。これはユーザへの負担となる。

- 並列実行ができない。
  連続して Ninf を用いた複数の計算をする場合を考える。複数の計算の間に依存関係がなければ、本質的に並列に実行することが可能である。しかし、通常のサーバ、クライアントモデルでは、サーバが単体であることを前提とするため、難しい。

これらの問題を解決するためには、何らかの機構がクライアントと Ninf サーバの中間に存在して、クライアントに対して、Ninf サーバの位置を隠蔽してくれたり、並列実行のマネージメントを行なってくれれば良い (図 4)。このための仕掛けとして、メタサーバを導入した。

3. メタサーバ

3.1 メタサーバの設計方針

メタサーバは、クライアントと Ninf サーバの仲介を行なう。このメタサーバ機構を設計する上では以下の点が重要である。

- クライアント、Ninf サーバを極力単純化する。
  メンテナンスを考慮すると、ユーザ、ライブラリ提供者がインストールする必要のあるこれらの部品は、単純でなければならない。
- ユーザの使用時の負担を最小にする。
  ユーザには、インストールの時を除いて、メタサーバの存在、場所を意識させたくない。このためには、メタサーバ機構の存在そのものをある程度、隠蔽しなければならない。

3.2 メタサーバアーキテクチャの設計

上記の要件をみたすようにメタサーバを以下の様に設計した。

- メタサーバは、クライアントからのリクエストを Ninf サーバに中継する。メタサーバには複数の Ninf サーバを登録することができ、メタサーバは、計算のリクエストを適切な Ninf サーバに対して委譲する。“適切” な Ninf サーバに関しては後で述べる。

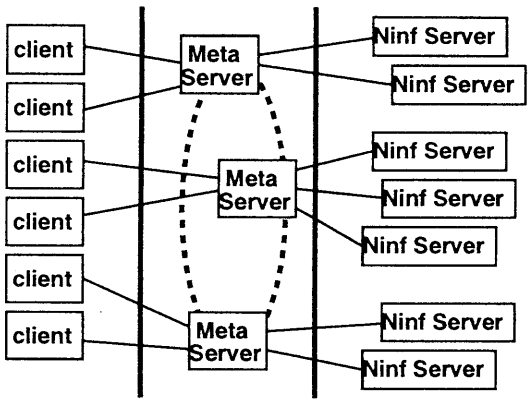

図5 メタサーバネットワーク
Fig. 5 Meta Server Network.

- メタサーバは他のメタサーバと通信し、情報を共有する。
  メタサーバを単体のサーバとして設計すれば、情報の管理は集中するため容易になるが、将来的にボトルネックになることは容易に想像がつく。複数のメタサーバを独立して立てることも考えられるが、新たに Ninf サーバを立てる際には、可能な限り多くのメタサーバに登録してもらわなければならない。これは繁雑である。これを避けるために、メタサーバ間での通信のプロトコルを作成し、メタサーバ間での情報の共有を可能にした。これにより、Ninf サーバを立てる際には、最寄りのメタサーバに対して登録を行なえば、その情報は自然に伝搬するため、世界中からアクセスされることが可能になる。
- 並列実行を実現するために、クライアントプログラムに計算のトランザクションという概念を導入する。
  トランザクションは、いくつかの連続する Ninf 計算の呼びだしであり、ユーザによって明示的に定義される。クライアントは、トランザクションを一括して、メタサーバに投げる。メタサーバはトランザクションを構成する計算を適宜スケジューリングし、Ninf サーバに計算を依頼する。トランザクションの設定はプログラムへのわずかな変更で行なえる。

メタサーバの構造を図 5に示す。

3.3 Ninf サーバの選択

メタサーバは複数の Ninf サーバを登録する。個々の Ninf サーバに関して以下のような情報を管理する。

- Ninf サーバの位置 (アドレスとポート番号)
- 処理できる計算の種類
- Ninf サーバへの距離 (通信のスループット)
- Ninf サーバの計算能力

クライアントからのリクエストを受けた際に、メタ

サーバは処理時間が最小になるように、Ninf サーバを選択する。処理時間は、転送時間と計算時間の和である。処理時間を最小にするためには、そのリクエストを処理することができるすべての Ninf サーバに対して、そのリクエストの計算にかかる転送時間と計算時間を見積もらなければならない。

転送時間を見積もるためには、メタサーバが それぞれの Ninf サーバに対する通信のスループットを知っている必要がある。転送量はリクエストの時点で正確に知ることができる。

また計算時間を見積もるためには、それぞれの Ninf サーバの計算能力とその計算自身の計算量がわからなければならない。計算能力は、Ninf サーバを立てる際に登録する。ここで問題になるのは、計算の計算量である。現在、計算量のオーダーをインターフェイス情報の一部としてユーザーが提供する方法を検討している。しかしオーダのみでは実際の計算量は類推できないので、過去の計算のログとオーダーから計算量を推定することを検討している。

3.4 メタサーバ間の通信

メタサーバは、最低でも一つの自分以外のメタサーバを知っている。メタサーバ間での情報の共有のための通信は、On demand で行なう。すなわち、メタサーバがリクエストを受けとった際に、そのリクエストを処理することができる Ninf サーバを知らなかった場合に、他のメタサーバに対して問い合わせを行なう。

この問い合わせを受けとった、メタサーバが適切な Ninf サーバを知らない場合には、さらに他のメタサーバへ問い合わせを forward する。このようにすることで、Ninf サーバは、最寄りのメタサーバに対して登録するだけで、他のメタサーバからも参照されることができる。

ここで問題になるのは、メタサーバが編目状のネットワークを構成することから、問い合わせが重複してしまったり、いつまでもネットワークの中を回り続けたりする可能性があることである。これに対応するために、問い合わせには、ユニークな ID とカウンタを設定した。メタサーバは、問い合わせの履歴を保持し、かつて処理した ID をもつ問い合わせは無視する。また、問い合わせを forward する際には、カウンタをデクリメントし、カウンタが 0 になった場合には、それ以上の forward を行なわない。これによって、問い合わせは失効するため、無限ループを構成することはない。

3.5 メタサーバによる分散並列実行

3.5.1 トランザクション

メタサーバによって、分散並列実行をするためには、計算のどの部分を並列実行して良いか、を何らかの方法で指定する必要がある。

このための概念としてトランザクションを導入した。トランザクションは、いくつかの ninf call をまとめたもので、トランザクション内の計算の順序はデータ依存関係を保持する限り、自由に変更しても構わない。

トランザクションはユーザが明示的に指定しなければならない。このために新たに二つのライブラリ関数をを用意した。`Ninf_check_in()` がトランザクションの開始を意味し、`Ninf_check_out()` はトランザクションの終了を意味する。この領域で囲まれた命令がトランザクションを構成する。トランザクションの開始と終了は、動的に行なわれるため、トランザクション内部に分岐などの構造を含むことも可能である。

3.5.2 分散並列実行

client プログラムでは、トランザクション内の ninf call 間の依存関係を解析して、データフローを作成する。client プログラムは、このデータフローをメタサーバに送る。メタサーバはそのデータフローに基づいて適切なスケジューリングを行なって、複数の Ninf サーバに対して個々の ninf call を行なう。このときに、データ依存関係がない ninf call に関しては並行して ninf call を発行することで、並列処理が実現される。

トランザクションを導入すると、計算の中間結果などの本質的にその後の計算に必要のない情報がわかる。このような値に関しては、メタサーバから、クライアントに書き戻さないことが可能である。これによって、通信コストの削減が可能である。

4. 実　　装

4.1 実装言語

メタサーバの実装には、Java[3] を用いた。Java は以下のような特徴をもち、メタサーバの実装に適している。

- bytecode interpreter であるため、移植性が高い。
- 容易に使用できる thread と、thread 間の同期機構を持つ。

4.2 メタサーバの実装

メタサーバは、ひとつのプロセスとして実装している。クライアントからリクエストを受けとると、新たに thread を作成してそのリクエストを処理する。他のメタサーバからの通信に対しても、同様に thread を作成して対応している。メタサーバ間の通信では メタサーバ内の情報が変化する場合があるが、メモリ空間を共有しているため、他の動作中の thread にも自動的に情報が伝搬するので、実装が容易である。

例として、クライアントから受けたリクエストを実行できる Ninf サーバを知らなかった場合を考えてみる。この場合、このリクエストを処理する thread は、他のメタサーバに問い合わせを発行する。この問い合わせの返答は、他の thread が受けとるので、元の thread は、適当な間隔をあけて Ninf サーバの入った構造体を polling するだけでよい。返答を受けとった thread

は、構造体に書き込む処理だけを行なう。このように、構造体を経由したthread 間の通信が行なえるため実装が非常に容易である。

4.3 分散並列実行の実装

4.3.1 クライアント

クライアントのライブラリにトランザクションを定義するための関数Ninf_check_in, Ninf_check_out を用意した。

例として行列A,B の積E とC,D の積F をさらに掛け合わせたものをG に入れる計算を考えてみる。これは以下のように書ける。

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

Ninf_check_in 後のNinf_call ではその場で実行を行なわずに、データフローだけの作成を行なう。最後のNinf_check_out の時点で、実際の計算を行なう。

各Ninf_call 間の依存関係は、共有する構造データと入出力モードによって判断する。構造データを共有するかどうかの判断は、構造データのアドレスの比較によって行なう。上の例では、E、F が共有されている。

4.3.2 トランザクションの実行

トランザクションは、以下のように処理される。

(1) トランザクションの内部(Ninf_check_in() の後) で、Ninf_call が呼び出されると、クライアントはデータフローを作成する。まず、メタサーバにコネクトし、そのNinf_call のインターフェイス情報を取得する。このインターフェイス情報に、各引数の入出力モードが含まれており、このモード情報と、引数の共有状態から各Ninf_call 間の依存関係を導く。

(2) Ninf_check_out() によってトランザクションが終了すると、データフローが完成する。この時点でクライアントはメタサーバに対してデータフローを送る。

(3) MetaServer では、受けとったデータフローを用いて、トランザクションに対応する疑似的なインターフェイス情報を作成しそれを送り返す。

(4) クライアントは、通常のNinf呼び出しとおなじ手順でそのインターフェイス情報に対して実行をおこなう。すなわち、そのインターフェイス情報にしたがって、引数のマーシャリングをおこない、送信する。また同様にして、結果の受信をおこなう。

(5) MetaServer はデータフローにしたがって、複数の関数を順次ディスパッチする。すべての関数の実行が終了すると、書き戻すべき情報をクライアントに送信して終了する。

疑似的なインターフェイス情報は、通常のNinf 呼びだしで用いているインターフェイス情報と同じ形式を用いている。このため、クライアントから見ると、疑似インターフェイス情報を受けとった以降の動作は、通常のNinf 呼び出しと何ら変わらない。このようにすることで、クライアントの構造を単純にすることができた。

4.3.3 疑似インターフェイス情報の実装

複合関数のインターフェイス情報 は基本的に、そのトランザクション内の個々のNinf 呼びだし のインターフェイス情報をすべて連結したものである。ただし、中間状態としてしか出現せず、書き戻す必要も送り出す必要もない引数(前出の例では E,F) に関しては、WORK というマークをつけて、余計な通信が生じないようにする。

前出の例で実例をみてみよう。もとのmmul の インターフェイス情報は、

```
[scalor:IN,array:IN,  array:IN,  array:OUT]
```

である。トランザクション全体としてのインターフェイス情報は、

```
[scalor:IN,array:IN,  array:IN,  array:WORK,
 scalor:IN,array:IN,  array:IN,  array:WORK,
 scalor:IN,array:WORK,array:WORK,array:OUT ]
```

となる。

4.3.4 並列実行制御

メタサーバからは、並列にディスパッチをかけるので、ディスパッチ間の同期をとる機構が必要である。この機構を Java の thread を用いて実現した。各関数実行に対して、それぞれ thread を割り当てる。また、データ依存関係に応じて各関数実行間に依存関係をつけ、それぞれ上流の thread がすべて終了するのをまって実行が開始されるようにした。

具体的には各 thread は上流の thread の数をカウンタとして持つ。各 thread は、自身が終了した時点で下流の thread にシグナルを送る。シグナルを受けとった thread はカウンタをデクリメントし、カウンタが0になった時点で実行を開始する。

すべての ninf call が終了した後、結果をクライアントへ書き戻す必要がある。このために書き戻し専用の thread を作成する。この thread はすべての ninf call を行なう thread の終了を待って実行される。

図6に上記の例の場合の thread の依存関係を示す。

5. 評　　価

5.1 評価環境

評価に用いたプログラムは非常に簡単な行列の積を求めるものである。実行は、50MHz の Sparc Server 2000 で行なった。対象とする行列は、double の 100 * 100 のものである。実行時間は 10 回の試行の平均である。

WORK は本来、計算のための作業領域を意味している。作業領域は転送が必要ないので転送されない。

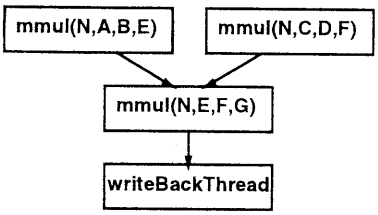

図6 Thread の依存関係
Fig. 6 Dependency among threads.

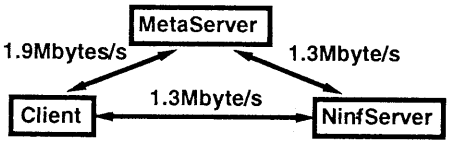

図7 評価環境の通信速度
Fig. 7 Evaluation Environment.

評価環境の通信速度を図7に示す。

5.2 Ninf Call によるオーバーヘッド

Ninf call を用いずに、行列を生成し計算した場合の実行時間は 2.19 秒であった。Ninf call を用いた場合には、3.26 秒となり Ninf を用いたことによるオーバーヘッドは 1.07 秒程度である。

通信されるデータ量は行列 3 つで 240,000bytes であり、スループットを考えると、0.2 秒程度かかると考えられる。残りの 0.8 秒ほどのオーバーヘッドは、コネクション、ライブラリ関数の fork, exec、データのセットアップなどによるものと考えられる。

5.3 メタサーバによる forwarding の負荷

メタサーバを介した呼び出しでは 3.90 秒かかった。メタサーバを介することによるオーバヘッドは、0.64 秒である。このうち転送によるオーバヘッドは同様に 0.2 秒ほどであると考えられる。残りの 0.4 秒は、コネクションなどによるものであろう。

6. 議　　論

6.1 情報の共有手法

現在の実装では、情報の共有は On demand で行なわれる。この手法では、あるリクエストの処理が可能な Ninf サーバを一つでも知っていると、他のメタサーバへの問い合わせが生じないため、情報の共有化が不十分になる可能性がある。

これに対応するために、個々のメタサーバが能動的に情報を発信する手法が考えられる。この場合、発信のタイミングの設定や、情報の伝搬の範囲の限定に関して良く検討する必要があるだろう。

6.2 他の研究と比較

動的にデータフローを形成して並列実行を行なう分散処理系の例として、Mentat[4] がある。Mentat では MPL なる C++ の拡張言語を用いる。この言語では特定のクラスのメソッドが自動的に分散処理の対象になるので、ユーザはこれを用いて分散処理部分を指定する。

この手法に比較して、現在の Ninf の分散実行方式は以下の特徴を持つ。

- 言語を新たに設定せずに、C 言語の範囲内でほぼ当価な計算構造を提供している。ユーザにとっては、言語を新たに習得する必要がなく連続性がよい。
- C 言語の範疇で行なっているため、以下のような問題がある。
  - トランザクション内での、ninf call は実際にはその場で実行されない。したがってトランザクション内で先行する ninf call の結果を参照しようとしても当然ながら、おかしな値しか得られない。これは、ninf call 以外の通常の計算に関する依存関係を検出する方法がないためである。
  - 構造データの共有をポインタの一致で検出する。ほとんどのケースでこれで十分であると思われるが、これでは不十分な場合も存在する。

  これらの問題点を解決するためには、コンパイラ自身に手を入れる必要があるが、実行すれば移植性の低下は避けられないと思われる。

7. おわりに

本稿では、Ninf のメタサーバアーキテクチャについてのべ、簡単な評価を示した。

今後は実装の完成度を向上させ、より厳密な評価を行なっていく予定である。

謝辞 本研究を進める上で討議いただいたお茶の水女子大学長嶋雲兵助教授に感謝する。本研究は工業技術院研究情報基盤整備研究開発制度 (RING-Program)「広域分散情報ベース構築ソフトウエア」の一環として行なわれた。

参 考 文 献

1) 関口智嗣, 佐藤三久, 長島雲兵: ネットワーク数値情報ライブラリ: - Ninf の設計 -, 情報処理学会研究報告 HPC, Vol. 94, No. 94-HPC-52, p. 127 (1994).
2) 中田秀基, 佐藤三久, 関口智嗣: ネットワーク数値情報ライブラリ Ninf のための RPC システムの概要, TR 95-28, 電子技術総合研究所 (1995).
3) Gosling, J. and McGilton, H.: *The Java Language Environment: A White Paper*, Sun Microsystems, Inc. (1995).
4) Grimshaw, A.S.: Easy to Use Object-Oriented Parallel Programming with Mentat, *IEEE Computer*, pp. 39–51 (1993).