

Ninf-G の使い方

この文書は、具体例を示しながら Ninf-G の利用方法を説明したガイドラインである。
4 章以降で説明されている Ninf-G の使い方については、本文書とともに提供されている
サンプルプログラムを参照されたい。

目次

1. 基礎知識

- 1.1 バッチシステム
- 1.2 Globus Toolkit
- 1.3 Ninf-G
 - 1.3.1 Ninf-G で何ができるか
 - 1.3.2 Ninf-G が提供していない機能

2. Ninf-G の使い方

- 2.1 用語の解説と定義
- 2.2 仮定環境
- 2.3 サンプルプログラム
- 2.4 Ninf-G ユーザが用意するファイルと設定

3. 環境設定

- 3.1 Globus Toolkit 環境設定
- 3.2 Ninf-G 環境設定

4. Ninf-G を利用する前のプログラム

5. 1 台のサーバに RPC をかけるプログラム

- 5.1 サーバセットアップ
- 5.2 クライアントセットアップ
- 5.3 プログラムの Ninf-G 化

6. 2 台のサーバに RPC をかけるプログラム

- 7. 1 度のジョブ起動要求で複数のジョブを実行する
- 8. Ninf-G リモートオブジェクト機能を使用する
- 9. MPI を使用する

10. FAQ

- Q1. サーバ名としてドメイン名は必須か.
- Q2. 実行エラーの原因調査やログ出力に関して知りたい.
- Q3. "GRAM Job failed" というエラーが発生する.
- Q4. サーバホームディレクトリのファイル `~/gram_job_mgr_[数字].log` は何か.
- Q5. Ninf-G を利用したサンプルプログラムは無いか.
- Q6. サーバでの計算が長時間になるがなにか注意することはないか.
- Q7. non-thread flavor と pthread flavor のどちらを利用すれば良いか.
- Q8. MDS とは何か.
- Q9. デバッグ方法を知りたい.
- Q10. ハートビート関連の Warning が出力された.
- Q11. Ninf-G には共有メモリは用意されているか.
- Q12. コンパイラやリンカの設定はできるか.
- Q13. ジョブがすぐに実行されない.
- Q14. workDirectory とは何か.

Q15. stdout, stderr がクライアントに転送されない.

Q16. ローカル LDIF ファイルの作成方法を知りたい.

1. 基礎知識

Ninf-G は複数のクラスタにより構成される計算グリッド上で動作するタスク並列アプリケーションの開発, 実行を支援するソフトウェアである. Ninf-G は Globus Toolkit をベースに開発されており, すべてのクラスタにはバッチスケジューラ, Globus Toolkit および Ninf-G などのソフトウェアが適切にインストールされ, 利用可能な状態になっている必要がある. ここではバッチスケジューラ, Globus Toolkit および Ninf-G の概要について説明する.

1.1 バッチシステム

バッチシステムはユーザによって queue に投入されたジョブを順次実行するシステムであるが, 特に以下のような機能を期待して導入されている:

- ✓ ユーザがどの計算ホストを利用するかを意識せずに(指定することなく), 自動的に空いているノードを選択してユーザのジョブを実行する.
- ✓ 多くのユーザがクラスタを共有する際に, 同時に実行可能なジョブの数を制御し, ユーザに仮想的な占有環境を提供したり, あるいは過負荷状態にならないようにする.
- ✓ 全てのユーザが必ずバッチシステムを介して利用することにより, クラスタの利用状況や計算ホストで動いているプロセスの情報はバッチシステムにより確認することができる. 計算ホストに変なプロセスが残るような状況を防ぐことができる.

これらは大規模クラスタの運用/利用においては重要であり, 実際, 大規模クラスタの多くはバッチシステムを導入し, 利用に際しては必ずバッチシステムを利用するよう利用方法を制限している.

バッチシステムとしては, Portable Batch System (PBS)や Sun Grid Engine(SGE) などが広く利用されている.

1.2 Globus Toolkit

Globus Toolkit はグリッドのソフトウェア, アプリケーションを開発, 実行するために必要な機能(の一部)を実装するための UNIX コマンドや C/Java の API を提供するソフトウェアパッケージである. 現在 Globus Toolkit の最新版は Version 4 であるが, Ninf-G Version 4 は Globus Toolkit の API を用いて実装されている. また, Pre-WS GRAM および WS GRAM を介した遠隔手続き呼び出しが可能である. サーバ側では利用するジョブ起動機構(Pre-WS GRAM and/or WS GRAM)が適切にインストールされている必要がある.

1.3 Ninf-G

Ninf-G はグリッドにおける遠隔手続き呼び出し(Grid Remote Procedure Call, GridRPC)によるプログラムの開発および実行を支援するソフトウェアである.

Globus Toolkit が提供する C/Java の API は非常に低レベルなものであり, それらを利用したアプリケーションの開発は非常に困難である. Ninf-G4 は Globus Toolkit の上位ミドルウェアとして, ユーザに対してはグリッドや Globus Toolkit が持つ複雑さを隠蔽し, グリッド上の分散計算資源を利用するアプリケーションの開発・実行を支援する. Ninf-G の詳細は本書第 3 章の「Ninf-G の使い方」および「Ninf-G Users' Manual」(<http://ninf.apgrid.org/documents/ng2-manual/user-manual.html>) を参照されたい.

1.3.1 Ninf-G で何ができるか

簡単に説明すると、Ninf-G を利用すると以下のことが可能となる。

- ✓ RPC モデルによるプログラムの並列化
- ✓ RPC モデルの利用によるグリッドプログラムの迅速な開発
- ✓ 複数の計算サーバを利用した大規模計算
- ✓ 計算サーバの動的な確保と利用，動的な解放
- ✓ 計算サーバやネットワークの故障/障害の検知と回避のできるプログラムの開発

1.3.2 Ninf-G が提供していない機能

簡単に説明すると、Ninf-G は以下の機能を提供していない。必要であれば、Ninf-G を利用するプログラムの側で実現すべき機能を以下に挙げる。

- ✓ スケジューリング機能(使用可能リソースの動的な検索や選択利用，計算サーバの負荷状況による動的な計算投入先の選択)
- ✓ 障害検知後の自動的な計算の再投入
- ✓ MPI プログラムを自動的に Ninf-G プログラムに変換する機能
- ✓ 計算プログラムの自動並列化
- ✓ 共有メモリモデルによる並列プログラミング
- ✓ 計算サーバ自体の利用許可申請を含む，自動的な計算サーバの発掘と利用

2. Ninf-G の使い方

4章以降では、用意したサンプルプログラムをもとに、実際に Ninf-G を利用する手順について解説する。まず Ninf-G を利用しない逐次版の C プログラムを起点に、そのプログラムを Ninf-G を利用するプログラムへと書き換え、徐々に利用方法を高度化する。

4章以降はサンプルプログラムを手元に置き、実際に手順を実行しながら確認する形で読んで頂きたい。

2.1 用語の解説と定義

この章以降で利用する用語を定義し、Ninf-G を利用する際に使用する用語を解説する。

- ✓ ユーザ
Ninf-G を利用したプログラムを作成する者をユーザとする。
- ✓ RPC 関数
Ninf-G では実際に数値計算を行う関数のことを指す。計算サーバ上で実行され、実行のためには計算サーバ上に計算リソース(CPU 計算時間，メモリ)を必要とする。RPC 関数はユーザが作成するが、他のユーザが作成した RPC 関数を利用することもできる(RPC 関数を共有する)。
(ユーザは計算リソースを必要とする関数を RPC 関数として設定する。)
- ✓ Ninf-G クライアント
RPC 関数を呼び出し、計算全体のコントロールを行うプログラムやプロセスのことを指す。Ninf-G クライアントプログラムはユーザが作成する。
- ✓ Ninf-G Executable, Ninf-G サーバプログラム
RPC 関数として数値計算を行うためのプログラムのことを指す。Ninf-G クライアントから関数ハンドル作成時に起動される。起動すると RPC 関数実行(計算)の開始要求を待ち受ける。

Ninf-G Executable には、1つの RPC 関数のみ定義可能であり、また、関数呼び出しの状態を保持しない(つまり、前回の関数呼び出しの結果は保持されていない)、ものと、複数の RPC 関数(メソッド)を定義することができ、また、関数呼び出しの状態を保持する(つまり、前回の関数呼び出しの結果が保持されている)ものの2種類がある。後者を Ninf-G リモートオブジェクトと呼ぶ。

- ✓ クライアントマシン
ユーザが Ninf-G クライアントプログラムを実行するマシン。
- ✓ サーバマシン
ユーザが Ninf-G サーバプログラムを実行するマシン。
- ✓ 関数ハンドル
Ninf-G クライアントが RPC 関数を呼び出すために準備/作成する `grpc_function_handle_t` 型のデータのことであり、Ninf-G Executable と Ninf-G クライアントとの接続(通信路)を抽象化したもの。

関数ハンドルを作成するには、計算サーバ名を与える。1つの関数ハンドルは計算サーバ上で起動された Ninf-G Executable プロセスと 1対1 対応する。

- ✓ オブジェクトハンドル
Ninf-G クライアントが Ninf-G リモートオブジェクトを作成し、メソッド関数を呼び出すために準備/作成する `grpc_object_handle_t_np` 型のデータのことであり、Ninf-G リモートオブジェクトと Ninf-G クライアントとの接続(通信路)を抽象化したもの。

オブジェクトハンドルを作成するには、計算サーバ名を与える。1つのオブジェクトハンドルは計算サーバ上で起動された Ninf-G Executable プロセスと 1対1 対応する。

- ✓ IDL ファイル
IDL とは、Interface Description Language の略であり、IDL ファイルには RPC 関数を呼び出すための入出力の型や並びを定義する。IDL ファイルはユーザが作成する。

Ninf-G IDL では、実際の計算処理手続きや、IDL に記述された関数を実行するためにリンクが必要なオブジェクトファイル(.o で終わるファイル)等の指定をすることもできる。

この、ユーザが作成した IDL ファイルを元に、Ninf-G サーバプログラムやローカル LDIF ファイルが作成される。

- ✓ ローカル LDIF ファイル
ユーザが作成した RPC 関数を呼び出すための、引数の並びなどの情報が記憶されたファイル。

IDL ファイルをサーバマシン上でコンパイルする際に作成され、クライアントコンフィギュレーションファイルから読み込んで使用する。

なお、これ以降コマンドラインの説明において "%" および "\$" にて始まる行がある。これは、それぞれシェル上のコマンドプロンプトを意味しており、"%" で始まる場合は csh (C シェル) や tcsh 上で実行していることを表す。"\$" で始まる場合は、sh (Brune シェル)

ル) や `bash` 上で実行していることを表す.

2.2 仮定環境

説明するにあたって, 利用する環境を以下に示す.

クライアントマシン : `client.example.org`
サーバマシン : `server01.example.org` から `server02.example.org`

上記のマシン名はそれぞれ仮定であり, 実在しない. 実際にサンプルを試す場合には, ユーザが利用するマシン名にそれぞれ読み替えた上で編集/実行して頂きたい.

2.3 サンプルプログラム

サンプルプログラムを下記に示す.

各サンプルが保存されているディレクトリ名を, ":" の右側に記述している.

章	内容	ディレクトリ名
4 章	Ninf-G を利用する前のプログラム	: <code>serial</code>
5 章	1 台のサーバに RPC をかける	: <code>1site</code>
6 章	2 台のサーバに RPC をかける	: <code>2site</code>
7 章	1 度のジョブ起動要求で複数のジョブを実行する	: <code>array</code>
8 章	Ninf-G リモートオブジェクト機能を使用する	: <code>object</code>
9 章	MPI を使用する	: <code>mpi</code>

2.4 Ninf-G ユーザが用意するファイルと設定

Ninf-G を利用するために, ユーザが用意するファイルには以下のものがある. サンプルプログラムには以下のものが含まれている.

- (1) サーバマシンに必要なファイル
 - ✓ IDL ファイル
 - ✓ ユーザが IDL ファイルにて指定した計算関数のオブジェクトファイル, および, ライブラリファイル
- (2) クライアントマシンに必要なファイル
 - ✓ Ninf-G クライアントプログラム
 - ✓ コンフィギュレーションファイル
 - ✓ ローカル LDIF ファイル(IDL コンパイル時に生成される)

サンプルプログラムを実行する際には, 以下の修正を行った上で実行する.

- ✓ クライアントコンフィギュレーションファイルの修正を行う.
ファイル中の "`example.org`" となっている箇所を, 実際に使用するサーバのホスト名に修正する.
- ✓ `<SERVER>` セクションの `hostname` を変更する.
- ✓ `<LOCAL_LDIF>` セクションの `filename` に含まれるホスト名部分を変更する.

3. 環境設定

Ninf-G を利用するためには, あらかじめ Globus Toolkit 及び, Ninf-G のユーザ環境設定を行う必要がある. 尚, この設定はログイン毎に必要となるが, シェルが起動毎に読み

込むファイル(.cshrc, .login や, .profile, .bashrc など)に手順を記述しておけば手動で設定する必要はなくなる.

3.1 Globus Toolkit 環境設定

Globus Toolkit のユーザ環境については, Globus Toolkit のマニュアルを参照されたい. 例えば Globus Toolkit Version 2 の場合, 以下の手順で設定する.

- ✓ 環境変数 GLOBUS_LOCATION を Globus Toolkit インストールディレクトリに設定する.
- ✓ 環境設定ファイル globus-user-env.{sh,csh} を読み込む.

(ユーザがシェルとして sh, bash を利用している場合の実行例)

```
$ GLOBUS_LOCATION=[Globus Toolkit インストールディレクトリ]
$ export GLOBUS_LOCATION
$. $GLOBUS_LOCATION/etc/globus-user-env.sh
```

(ユーザがシェルとして csh, tcsh を利用している場合の実行例)

```
% setenv GLOBUS_LOCATION [Globus Toolkit インストールディレクトリ]
% source $GLOBUS_LOCATION/etc/globus-user-env.csh
```

3.2 Ninf-G 環境設定

Ninf-G のユーザ環境は, 以下の手順で設定する.

- ✓ 環境変数 NG_DIR を, Ninf-G インストールディレクトリに設定
- ✓ 環境設定ファイル \$NG_DIR/etc/ninfg-user-env.{sh,csh} 読み込み

(ユーザがシェルとして sh, bash を利用している場合の実行例)

```
$ NG_DIR=[Ninf-G インストールディレクトリ]
$ export NG_DIR
$. $NG_DIR/etc/ninfg-user-env.sh
```

(ユーザがシェルとして csh, tcsh を利用している場合の実行例)

```
% setenv NG_DIR [Ninf-G インストールディレクトリ]
% source $NG_DIR/etc/ninfg-user-env.csh
```

4. Ninf-G を利用する前のプログラム

Ninf-G を利用する前のサンプルプログラムを serial ディレクトリに用意した.

このサンプルプログラムは, モンテカルロ法により円周率 π の値を計算し求める簡易的なプログラムである. モンテカルロ法では, まず長さ 1 の正方形中にランダムな点を打つ. 次にその点の原点からの距離が 1 以内かどうか調べる. そして, 同じように何度も点を打ち距離を計測することによって円周率を求める.

円周率は, 以下の式で求めている.

$$\text{円周率} = \text{原点からの距離が 1 以内だった点の数} \div \text{打った点の総数} \times 4$$

この方法では, 点を打った回数によって, 求める円周率の精度が変化する. 求める円周率の精度を向上させたい場合, 点を打つ回数を多くすればよい.

点を打つためには、多少の計算能力が必要であり、そのためより精度の良い結果を得るためには、より多くの計算能力が必要である。そして、計算能力をより多く利用する手段として、Ninf-G では RPC を用意している。

Ninf-G を利用する前のプログラム(pi_serial.c)では、単純に実行した場合、計算能力、計算リソースとして 1つの CPU しか利用することはできない。しかし、Ninf-G を利用すると、複数の計算サーバの CPU を同時/並列に利用することで、同じ計算時間内によりたくさんの点を打つことができるようになる。その結果、精度の良い計算結果が得られるようになる。

このプログラムに利用されている pi_trial() 関数のように、計算結果を得るためには、入力として与えられた情報のみあればよく、後は計算するのみの関数が一般的には RPC として適している。

以下の手順で、プログラムを実行する。

(1) ディレクトリの移動

```
% cd serial
```

(2) コンパイル

```
% make
```

実行ファイル "pi_serial" が作成される。

(3) 実行

```
% ./pi_serial 100000
```

結果、計算された円周率の値が表示される。100000 は打つ点の数であり、任意の数字を入力する。

5. 1 台のサーバに RPC をかけるプログラム

まずは、最も簡単な 1 台のサーバに RPC をかけるプログラムを用いる。

クライアントプログラムから 1 台のサーバに対して RPC を実行する。

サンプルプログラム : lserver

サーバマシン : server01.example.org (手順 5.1)

クライアントマシン : client.example.org (手順 5.2)

下記実行前にサンプルのディレクトリに移動する。

```
% cd lsite
```

5.1 サーバセットアップ

(1) IDL ファイルのコンパイル

```
server01% ng_gen pi.idl
```

(2) Ninf-G Executable 作成

```
server01% make -f pi.mak
```

5.2 クライアントセットアップ

(1) クライアントプログラムのコンパイル

```
client% ng_cc -o pi_client_1server pi_client_1server.c
```

(ここまでの手順(サーバセットアップ (1), (2), および, クライアント
セットアップ (1))のルールはサンプルプログラムの Makefile に
記述されている. そのため, make コマンドの実行で代用できる.)

(2) Local LDIF ファイルを server01 から client にコピーする

```
client% scp server01.example.org:samples/1server/pi.server01.examples.org.ngdef .
```

(3) コンフィギュレーションファイルの修正

client.conf ファイルを編集する.

(vi や emacs コマンド等のテキストエディタを利用する.)

→ "example.org" を "server01.example.org" に修正する.

→ "pi.example.org.ngdef" を "pi.server01.example.org.ngdef" に修正
する.

(4) クライアントプログラムの実行

```
% grid-proxy-init  
'パスフレーズ入力'
```

(なお, grid-proxy-init コマンドはクライアントの実行毎に毎回必要
となるわけではない. 作成した一時証明書の有効期限が切れるまでは,
grid-proxy-init コマンドを再度実行する必要はない)

```
% ./pi_client_1server 10000 server01.example.org
```

pi_client_1server は「打つ点の数」と「サーバホスト名」を引数で受け取る.

ここで注意するのは, Ninf-G が利用する Globus Toolkit の特性として, ジョブの
起動 1 回につき数秒以上の時間が必要となるということである. Ninf-G としてはジョブ
の起動は, 関数ハンドル作成に対応しており, 関数ハンドルの作成に数秒以上の時間が必要
である.

そのため, 実際の RPC 計算は 1 秒以下であった場合でも, クライアントプログラ
ムの実行には数秒以上の時間がかかってしまうことになる.

また, この待ち時間はジョブのバッチシステム, キューイングシステムによっても
変化する. バッチシステムが行うスケジューリング次第で, ジョブの起動がすぐには始ま
らず, 他のジョブの終了まで待たされる可能性もある.

5.3 プログラムの Ninf-G 化

この章で実行した 1server/pi_client_1server.c は, 先ほど 4 章で説明したプログラムを
Ninf-G 化したものである.

PI を計算する Ninf-G サーバプログラムは IDL ファイルである pi.idl の中で定義され
ている. 具体的には pi_trial0 関数を RPC 関数とし, Ninf-G を経由して実行するよう
に変更した.

それに伴い, pi_trial0 関数自体は, IDL ファイルから呼び出すように変更し, その IDL
で定義した RPC 関数 pi_trial0 を Ninf-G クライアントプログラムから呼び出してい
る.

Ninf-G クライアントプログラムでは, 定義した RPC 関数 pi_trial0 を呼び出すための

手続きを追加している。追加した手続きは GridRPC API である以下の関数を呼ぶものである。

- ✓ Ninf-G を利用するための初期化/終了 API
 `grpc_initialize()`, `grpc_finalize()`
- ✓ RPC を呼び出すための関数ハンドルの作成と破棄用 API
 `grpc_function_handle_init()`, `grpc_function_handle_destruct()`
- ✓ 関数ハンドルに対し、RPC の実行を要求する API
 `grpc_call()`
- ✓ RPC 実行の終了と結果を待つ API
 `grpc_wait_all()`

この GridRPC API を利用することで、Ninf-G を利用し RPC モデルによる計算ができるようになった。

6. 2台のサーバに RPC をかけるプログラム

次に、2台の計算サーバを利用して並列に RPC を呼ぶテストを行う。このプログラムが1台のサーバに対して RPC をかける5章のプログラムと異なる点は、

- ✓ 非同期呼び出しを用い、2台のサーバで並列に計算が進むようにする。
- ✓ 一方のサーバに対しては WS GRAM 経由で RPC をかける

の2点である。

1つのクライアントプログラムで2つのサーバプログラムを実行し、RPC を複数の計算サーバに振り分けることにより、並列に計算を行うことができる。

```
サンプルプログラム : 2servers
サーバマシン 1      : server01.example.org (手順 6.1)
サーバマシン 2      : server02.example.org (手順 6.1)
クライアントマシン : client.example.org   (手順 6.2)
```

下記実行前にサンプルのディレクトリに移動する。

```
% cd 2servers
```

6.1 サーバセットアップ

それぞれのサーバマシンにログインし、5.1 の手順と同様の操作をそれぞれ行う。

6.2 クライアントセットアップ

(1) クライアントプログラムファイルの修正

クライアントマシンにログインし、5.2 の手順と同様の操作を行う。

(2) クライアントプログラムのコンパイル

```
% ng_cc -o pi_client_2servers pi_client_2servers.c
```

(3) ローカル LDIF ファイルのコピー

`server01.example.org`, および, `server02.example.org` 上の
Ninf-G Executable をコンパイルしたディレクトリにある,

"pi.server01.example.org.ngdef", および,
"pi.server02.example.org.ngdef"を client.example.org にコピーする.

```
% scp server01.example.org:samples/2servers/pi.server01.example.org.ngdef .  
% scp server02.example.org:samples/2servers/pi.server02.example.org.ngdef .
```

(4) コンフィギュレーションファイルの修正

client.conf ファイルを編集する.

→ "example1.org" を "server01.example.org" に修正する.

→ "example2.org" を "server02.example.org" に修正する.

server02 に対しては

invoke_server GT4py

jobmanager jobmanager-sge

が指定されていることにより、WS GRAM 経由で RPC が行われる。

(5) クライアントプログラムの実行

```
% grid-proxy-init
```

```
'パスフレーズ入力'
```

```
% ./pi_client_2servers 10000 server01.example.org server02.example.org
```

pi_client_2servers は「打つ点の数」と「サーバホスト名」を引数で受け取る。「サーバホスト名」は複数指定可能で、引数で指定されたすべてのホストで計算を実行する。

この手順により、複数の計算サーバを用いて計算を並列に行うことができた。このプログラムの場合、コマンドライン引数として与えられた「打つ点の数」を、それぞれの計算サーバで分割して並列に計算するようにプログラムされている。

そのため、このプログラムでは 1 つのサーバを利用して計算する場合よりも、2 つのサーバを利用して計算した場合の方が計算の終了が速い結果となる。「打つ点の数」が 10000 程度であれば、すぐに終了して並列の効果は分からないが、1 つのサーバで計算した場合の計算時間が 1 日などの長い時間である場合複数のサーバを利用する効果は顕著に現れる。そして、利用する計算サーバの数は多ければ多いほど計算時間は短くなる。

7.1 度のジョブ起動要求で複数のジョブを実行する

通常、1 つのクラスタシステムは、複数の計算サーバ(計算ノード)を持っている。そして、クラスタシステム(ジョブ投入ノード)に対し投入されたジョブを、順次別々の計算ノードに割り振った上で実行する機能を有する。

Ninf-G から、それら計算ノードのいくつかを利用するためには、利用する数だけ関数ハンドルを作成する必要がある。

しかし、関数ハンドルの作成は 1 回実行するために最低でも数秒の時間が必要となってしまうため、複数の関数ハンドルを作成すると比例して時間がかかってしまう。また、関数ハンドル作成要求をするたびに、入り口ホストにジョブマネージャプロセスが起動されるため、数十あるいは数百を越える数の関数ハンドルを単純な方法(関数ハンドルを 1 つずつ作成する方法)で作成すると、入り口ホストが過負荷状態となってしまう。そこで、Ninf-G には一度の関数ハンドル作成要求で、複数の関数ハンドルを一度に作成する機能が用意されている。この機能を利用すると、関数ハンドルの作成時間を短縮するとともに、大量の(数十～数百の)関数ハンドルを作成することが可能となる。

この章ではこの機能を利用したテストを行う。

サンプルプログラム : array
サーバマシン : server01.example.org (手順 7.1)
クライアントマシン : client.example.org (手順 7.2)

下記実行前にサンプルのディレクトリに移動する
% cd array

7.1 サーバセットアップ

サーバマシン(クラスタ管理ノード)にログインし, 6.1 の手順と同様の操作を行う。

7.2 クライアントセットアップ

(1) クライアントプログラムのコンパイル

```
% ng_cc -o pi_client_array pi_client_array.c
```

(2) ローカル LDIF ファイルのコピー

server01.example.org 上の Ninf-G Executable をコンパイルしたディレクトリにある, "pi.server01.example.org.ngdef", をマシン client.example.org にコピーする。

```
% scp server01.example.org:samples/array/pi.server01.example.org.ngdef .
```

(4) コンフィギュレーションファイルの修正

client.conf ファイルを編集する。

→ "example.org" を "server01.example.org" に修正する。

→ "pi.example.org.ngdef" を

"pi.server01.example.org.ngdef" に修正する。

(5) クライアントプログラム実行

```
% grid-proxy-init
```

'パスフレーズ入力'

```
% ./pi_client_array 10000 server01.example.org 16
```

pi_client_array は「打つ点の数」・「サーバホスト名」・「ハンドル数」を引数で受け取る。

コマンドライン最後の 4 は, 作成する関数ハンドルの数である。例として 16 を使用したが, 16 である必要はない。利用しているクラスタシステムが, ほかのユーザと共同利用している場合, 作成するハンドルの数には注意し, 許可されている範囲内で使用するよう気をつけること。

このプログラムでは関数ハンドル作成時のオーバーヘッドを削減するとともに, 入り口ホストで起動されるジョブマネージャの数を少なく保つ「関数ハンドル一括生成機能」を利用している。これは大規模クラスタシステム(数十~数千プロセッサ規模)を利用するためには必須な技術である。

8. Ninf-G リモートオブジェクト機能を使用する

Ninf-G Executable は状態を持たない(state less)。つまり, 同じ関数ハンドルを用いて繰り返し RPC 関数を呼び出したとしても, 前回の呼び出し時の状態は Ninf-G Executable

には保持されていない。そのため、同じデータに対して何度か計算を行なうような場合でも毎回データを送信する必要がある。

Ninf-G2 は状態を保持する事を可能にした Ninf-G Executable をリモートオブジェクトとして提供している。リモートオブジェクトには複数の関数(メソッド)を定義することができ、Ninf-G クライアントがそれらのメソッドを呼び出すための機能を提供している。

リモートオブジェクト機能を利用する場合は、ハンドルは関数ハンドルではなくオブジェクトハンドルを作成する。オブジェクトハンドルは `grpc_object_handle_t_np` 型のデータであり、リモートオブジェクトと Ninf-G クライアントとの接続(通信路)を抽象化したものである。オブジェクトハンドルを作成した後、破棄するまでは対応するリモートオブジェクトが状態を保持することができる。そして、そのオブジェクトハンドルに対し、状態を変更する様々なメソッドを呼ぶこともできる。

例えば、RPC の入力データが非常に大きく、かつ各 RPC 呼び出し間で入力データが全く同じ場合には、リモートオブジェクト機能が有効に利用できる。各ハンドル作成後に 1 度だけその大きな入力データを渡す初期化メソッドを呼び出し Ninf-G Executable に保存するようにすれば、それ以降の各 RPC 呼び出しではその大きな入力データの転送が不要となり性能が改善される。

別の例として、1つの Ninf-G クラスに以下のメソッドを実装する例を挙げる。

- ✓ 初期化処理メソッド
引数を与える。その与えられた引数は、データ保持領域に保存する。
- ✓ 計算 1 メソッド
保持されたデータを使用して計算 1 を実行する。得られた途中結果は、データ保持領域に保存する。結果は返さない。
- ✓ 計算 2 メソッド
保存された途中結果より、計算 2 を実行する。計算 2 によって得られた最終結果をクライアントに返す。

サンプルプログラム : `object`
サーバマシン : `server01.example.org` (手順 8.1)
クライアントマシン : `client.example.org` (手順 8.2)

下記実行前にサンプルのディレクトリに移動する。
`% cd object`

8.1 サーバセットアップ

(1) IDL ファイルのコンパイル
`% ng_gen pi_object.idl`

(2) Ninf-G Executable 作成
`% make -f pi_object.mak`

8.2 クライアントセットアップ

(1) クライアントプログラムのコンパイル
`% ng_cc -o pi_client_object pi_client_object.c`

(2) ローカル LDIF ファイルのコピー
`server01.example.org` 上の Ninf-G Executable をコンパイルした

ディレクトリにある, "pi.server01.example.org.ngdef",
をマシン client.example.org にコピーする.

```
% scp server01.example.org:samples/object/pi_object.server01.example.org.ngdef .
```

- (3) コンフィギュレーションファイルの修正
client.conf ファイルを編集する.
→ "example.org" を "server01.example.org" に修正した.
→ "pi_object.example.org.ngdef" を
"pi_object.server01.example.org.ngdef" に修正した.

- (4) クライアントプログラムの実行

```
% grid-proxy-init  
'パスフレーズ入力'
```

```
% ./pi_client_object 10000 server01.example.org
```

pi_client_object は「打つ点の数」と「サーバホスト名」を引数で受け取る.

9. MPI を使用する

Ninf-G は MPI で書かれた並列プログラムを RPC 関数として設定することができる.
この機能を利用することにより, 今まで述べてきたマスター/ワーカ型に基づくタスク並列
的なプログラミングの他に, 遠隔手続き呼び出しを行なったサーバマシン上で MPI によ
る細粒度並列処理を行なうプログラミングも可能となる.

サンプルプログラム : mpi
サーバマシン : server01.example.org (手順 9.1)
クライアントマシン : client.example.org (手順 9.2)

下記実行前にサンプルのディレクトリに移動する

```
% cd mpi
```

9.1 サーバセットアップ

- (1) IDL ファイルのコンパイル

```
% ng_gen pi_mpi.idl
```

- (2) Ninf-G Executable の作成

```
% make -f pi_mpi.mak
```

9.2 クライアントセットアップ

- (1) クライアントプログラムのコンパイル

```
% ng_cc -o pi_client_mpi pi_client_mpi.c
```

- (2) ローカル LDIF ファイルのコピー

server01.example.org 上の Ninf-G Executable をコンパイルした
ディレクトリにある, "pi_mpi.server01.example.org.ngdef",
をマシン client.example.org にコピーする.

```
% scp server01.example.org:samples/mpi/pi_mpi.server01.example.org.ngdef .
```

- (3) コンフィギュレーションファイルの修正

client.conf ファイルを編集する。
→ "example.org" を "server01.example.org" に修正した。
→ mpi_runNoOfCPUs の値を、使用する CPU 数に調整する。
→ "pi_mpi.example.org.ngdef" を
"pi_mpi.server01.example.org.ngdef" に修正した。

(5) クライアントプログラムの実行

```
% grid-proxy-init  
  'パスフレーズ入力'  
% ./pi_client_mpi 10000 server01.example.org
```

pi_client_mpi は「打つ点の数」と「サーバホスト名」を引数で受け取る。ここで使用する CPU の数は、(3) で示した通りコンフィギュレーションファイルの SERVER セクションの mpi_runNoOfCPUs で指定している。他にも Ninf-G では MPI の CPU 数を指定する方法がいくつかあるが、別の方法については Ninf-G ユーザーズマニュアルを参照されたい。

MPI は既存の並列システムにおける並列プログラミングモデルとして有名であり、MPICH-G2 などのグリッド対応の MPI 実装を用いることにより、既存の MPI プログラムをそのままグリッド上で実行できるという利点がある。その一方、グリッドにおいては、MPI が必要とする co allocation (プログラム実行開始時に、すべての MPI プロセスが起動されなければならない)を保証することは非常に難しい。また、process spawning などの機能を使えばある程度対応可能であるが、基本的には MPI は利用するプロセッサ、プロセス数などが静的に決定されるものであり、グリッドが持つ動的な性質とは相性が悪い。さらに、大規模環境で大規模アプリケーションを長時間実行する事はグリッドの最大の魅力の一つであるが、そのような状況では計算ホストのハードウェア障害など、障害発生時にどのように対処できるかといった点が重要になる。しかし、MPI はどれか 1 つのプロセスに障害が発生した場合、その時点で全体のプログラムの実行が中止されてしまうなど、耐障害性の機能が弱い。

GridRPC は上記に示した問題点に対応可能なプログラミングモデルである。co allocation は必ずしも必要ではなく、関数ハンドルを動的に生成/破棄することによる動的な資源の追加/解放が容易である。また、どこかの計算ホストで障害が発生した場合にも他の計算ホストで行なわれている計算には影響せず、また、その計算ホストでの計算を破棄する、あるいは再度依頼するなどの形で障害発生時の対応を容易に実現できる。

MPI は、並列に計算を行う場合に人気のあるプログラミングモデルである。しかし、現状では必ずしも MPI の計算モデルに適していないプログラムまでもが MPI で記述されていることも多い。実装するアプリケーションの性質に応じて、最適なプログラミングモデルが選択されるべきであり、「独立した 1 つ以上の計算をグリッド上の分散資源で実行」するタイプであれば、(少なくとも MPI ではなく) GridRPC を使って実装されるべきである。また、ここで述べた「詳細で密に通信する必要のある計算の部分は MPI が担当し、通信が疎でよい部分は Ninf-G を利用する」という GridRPC と MPI を組み合わせる方法を用いることにより、双方のプログラミングモデルの利点を活かした大規模アプリケーションを実装することが可能である。

10. FAQ

過去に質問された項目や、注意した方がよい項目などを以下に挙げる。

Q1. サーバ名としてドメイン名は必須か。

Q. grpc_function_handle_init() API により関数ハンドルを作成する時には

サーバ名が必要だが、このサーバ名はドメイン名を省略してもよいか?
それともドメイン名を付ける必要があるか?

A. ドメイン名は必須である。

関数ハンドル作成時やクライアントコンフィギュレーションファイル作成時に、サーバ名は **FQDN (Fully Qualified Domain Name)** にて指定する。また、"localhost" は指定できない。正確に **FQDN** を指定しない場合、**Ninf-G API** がエラーで終了する。

例えば、サーバとして **server.example.org** を利用する場合、"server" とは指定できない。必ず **"server.example.org"** と指定する。

Q2. 実行エラーの原因調査やログ出力に関して知りたい。

Q. **Ninf-G** を利用するクライアントプログラムを作成し実行したが、うまく実行されていないようだ。 **Ninf-G API** の実行に失敗しているようである。何か原因の追求方法は無いか?

A. **Ninf-G** には、カスタマイズ可能なログ出力の機能がある。クライアントコンフィギュレーションファイルや、サーバのコンフィギュレーションファイルにログ関連の設定をすることで、 **Ninf-G API** 実行時のエラー情報やデバッグ情報等を出力させることができる。出力先はファイルへ設定することもできる。

このログ機能を利用すると、 **Ninf-G API** 実行の状態を確認することが可能であり、 **API** 実行の失敗原因をトレースすることができる。

ログ機能設定の詳細は、 **Ninf-G ユーザーズマニュアル**を参照されたい。
<http://ninf.apgrid.org/documents/ng2-manual/user-manual.html>
(クライアントのログ出力設定 : 4.3.9 節)
(サーバのログ出力設定 : 3.3.2 節)

ログ出力メッセージは、それぞれ以下の順で出力される。

"日付 時刻:クライアント/サーバ:ホスト名:ログレベル:

各データ構造の ID:Ninf-G 内部関数名:ログメッセージ"

また、 **Ninf-G** のすべての **API** は実行の成功、失敗を表す **grpc_error_t** 型の値を返す。これらの値は、ユーザプログラムで必ずチェックするようプログラムを記述した方がよい。

Ninf-G Executable にて実行される計算関数が **Segmentation fault** を起こしているような場合には、クライアントコンフィギュレーションファイルの **<SERVER>** セクション **coreDumpSize** 属性が有効な場合もある。

Q3. "GRAM Job failed" というエラーが発生する。

Q. **grpc_call()** や **grpc_call_async()** がエラーで終了する。この原因は何か? エラー発生時のエラーのログを出力すると、
"ngctlJobCallback: GRAM Job failed because ..." といったログメッセージが出力されている。

A. "GRAM job failed" といったエラーが出力される場合には、 **Ninf-G** が利用

している Globus Toolkit のジョブ実行モジュールである GRAM がエラーとなっている。

GRAM がエラーとなる原因には様々なものがあり、一概に判別することはできない。エラーメッセージ毎に対処方法は異なる。

あるサーバの GRAM が正常に利用できるかを判断するには、コマンドラインから以下のコマンドを実行する。

```
% globus-job-run server.example.org /bin/hostname
```

上記コマンドは、server.example.org の GRAM に対し、/bin/hostname コマンドの実行を要求する。成功した場合、ホスト名が表示される。このコマンドを実行することにより、問題の解決となる場合がある。

GRAM の失敗について、Globus の Web ページを参照されたい。

また、Ninf-G クライアントのログレベルを Information や Debug へと設定することで、クライアントが GRAM へ渡した GRAM RSL (Resource Specification Language: 実行環境指定言語)を確認することもできる。

Q4. サーバホームディレクトリのファイル ~/gram_job_mgr_[数字].log は何か。

Q. Ninf-G を利用した際、サーバのホームディレクトリに gram_job_mgr_[数字].log というファイルが作成されていた。このファイルは何か?

A. このファイルは、Ninf-G が利用している Globus Toolkit のジョブ実行モジュールである GRAM の出力するログファイルである。

GRAM は、ジョブを実行すると jobmanager プロセスを同時に起動する。jobmanager プロセスはジョブを監視し、終了の検知や、強制終了などを行う。

jobmanager はジョブが正常終了した場合は、ログファイルを消去するが、何らかのエラーが発生したり、ジョブの実行がキャンセルされた場合には、ログファイルを消去しない。

そのため、ホームディレクトリに GRAM のログファイルが残ることとなる。

Ninf-G クライアントの場合、grpc_function_handle_init() API により関数ハンドルを作成したあと、grpc_function_handle_destruct() にて関数ハンドルを破棄する前に、クライアントを ^C キーにより、強制終了した場合などに ハンドルに該当するジョブがキャンセルされる。

そのため結果的に、正常終了しなかった場合にはサーバのホームディレクトリに gram_job_mgr-[数字].log というファイルが残ってしまう。

ジョブ終了の原因を追求する必要がなければ、このファイルは削除してもよい。

Q5. Ninf-G を利用したサンプルプログラムは無いか。

Q. Ninf-G を利用したプログラムを作成しようと思うが、書き方が分からない。サンプルプログラムは無いのか？

A. Ninf-G パッケージには、Ninf-G を利用したサンプルプログラムが用意されている。

Ninf-G は、Ninf-G ダウンロードの Web ページから得ることができる。
<http://ninf.apgrid.org/packages/welcome.shtml>

Q6. サーバでの計算が長時間になるがなにか注意することはないか。

Q. 関数ハンドルを作成した後、数日以上そのハンドルを利用する予定である。この際気をつけることはないか？

A. 一時証明書の有効期限に注意すること。grid-proxy-init コマンドは、無指定の場合の有効期限を 12 時間として一時証明書を生成する。

12 時間以上関数ハンドルを利用したい場合は、2 つの方法がある。

1. grid-proxy-init コマンドの -valid オプションを利用する。
2. Ninf-G の Refresh Credentials 機能を利用し、定期的に一時証明書を再作成しながら計算を継続する。

Q7. non-thread flavor と pthread flavor のどちらを利用すれば良いか。

Q. Globus Toolkit には non-thread flavor と pthread flavor が用意されており、Ninf-G はどちらでもコンパイル可能である。どちらを利用すれば良いか？

A. pthread flavor を利用することをお勧めする。Ninf-G クライアントには pthread flavor にてコンパイルしなければ利用できない機能があるため、クライアントは pthread flavor を利用した方がよい。

しかし、その機能が必要なければどちらの flavor を利用しても問題ない。また、Ninf-G Executable も同様、どちらを利用しても問題ない。

もし、ユーザが記述するプログラムが Pthreads を利用したプログラムであれば、Ninf-G も pthread flavor を利用する必要がある。

尚、Globus Toolkit は non-thread flavor, pthread flavor のどちらでも仕様上はほぼ同じ機能を提供するが、内部実装が全く異なるため、利用する flavor によって性能の差が発生する。

Q8. MDS とは何か。

Q. Ninf-G は MDS を利用するようである。MDS とは何か？ 利用する必要はないのか。

A. MDS とは、Globus Toolkit が提供する情報サービスである。Ninf-G は、MDS を利用することができる。MDS を利用して、計算サーバにインストー

ルされた各 RPC の情報を得ることができる。

しかし、この情報はローカル LDIF ファイルを利用しても得ることができるため、必ずしも MDS を利用する必要はない。

尚、MDS を利用する場合には以下の点に注意が必要である。

- MDS は、Globus Toolkit ディレクトリに、Ninf-G RPC 関数情報をインストールして利用する。Globus Toolkit ディレクトリ以下の `var/gridrpc` に、(MDS の内部実装である LDAP が参照するための) LDIF ファイルをインストールすることで情報が提供される。しかし、MDS はキャッシュ機能を持っており、更新には 10 分程度の時間がかかる場合がある。そのため、LDIF ファイルインストール直後は情報検索に失敗する時がある。

尚、どのような RPC がインストールされているかは、Ninf-G の `ng_dump_functions` コマンドを利用して参照することができる。

- MDS を利用するために LDIF ファイルをインストールするディレクトリは、共有ディレクトリである。個人個人で別のディレクトリが用意されているわけではない。そのため、同じ計算サーバ上に、同じ名前の RPC 関数を複数の人が同時に作成し、インストールすることはできない。

ディレクトリが共有されているため、他のユーザの迷惑とならないよう注意して利用する必要がある。

- `VO_NAME` をあらかじめ知っておく必要がある。
MDS にて検索する場合には、情報を要求したいサーバに設定されている `VO_NAME` を知っておく必要がある。`VO_NAME` を知らなければ検索はできない。もし、MDS サーバにログインが可能であれば、Globus Toolkit ディレクトリ以下の `etc/grid-info-slapd.conf` を参照すると `VO_NAME` が分かる可能性が高い。

Q9. デバッグ方法を知りたい。

Q. Ninf-G Executable はリモートで起動されるため、デバッグが難しい。
なにか方法はないか？

A. Ninf-G には、Ninf-G Executable のデバッグ方法が用意されている。
クライアントコンフィギュレーションファイルにて、`<SERVER>` セクションには、`debug_busyLoop`, `debug`, `coreDumpSize` が設定できる。詳細はユーザーズマニュアルを参照されたい。

また、Ninf-G のログ出力機能もデバッグ時には有効である。クライアントと Ninf-G Executable 双方ともログの出力を行うことができ、Ninf-G API の実行状況を逐一確認することができる。

Q10. ハートビート関連の Warning が出力された。

Q. Ninf-G クライアントのログ出力として、以下のようなメッセージが出力された。問題はないか？

"... heartbeat timeout warning (60 seconds) occurred ..."

A. 必ずしも問題が発生しているとは言えない。

ハートビートとは、Ninf-G Executable プロセスが正常に動作しているか、フリーズしていないか、ネットワークに問題がないかを検知するための Ninf-G の機能である。

Ninf-G Executable は、クライアントに対し定期的にハートビートを送信している。そのためもし、ネットワークがフリーズしていたり、Ninf-G Executable がフリーズしている場合は、ハートビートが Ninf-G Executable からクライアントへ届かなくなる。

ハートビートが Ninf-G クライアントへ一定時間届かなかった場合、クライアントはその Ninf-G Executable (関数ハンドル)を、「異常状態であり利用できなくなったもの」と判断し、エラーとする。その関数ハンドルはもう利用できない。実行中の RPC はエラーで返る。

Ninf-G クライアントはこの機能によって、フリーズすること無く障害の発生を検知することができる。そのため、ユーザは故障や障害に強いプログラムを作成することが可能である。

Ninf-G Executable がハートビートを送信する間隔を 60 秒とした場合、ハートビートの Warning が出力されるのは、Ninf-G クライアントに 60 秒間ハートビートが届いていない場合である。そのため、Ninf-G Executable の処理が少々遅滞し、ハートビートの定期的な送出手が少し遅れてしまった場合にも Warning は出力される。この状態はまだ異常であるとは断言できない。

デフォルトの設定では、ハートビートの送信間隔は 60 秒であり、ハートビートを受信しなかった場合に異常だと判断するタイミングは、 $60 \text{ 秒} \times 5 \text{ 回} = 300 \text{ 秒}$ の間ハートビートを受信しなかった場合となっている。(詳細はユーザーズマニュアルを参照されたい。)

なお、ハートビート Warning が出力された後、以下のメッセージが出力された場合は、ハートビートの送信が正常に戻ったことを意味する。

"... heartbeat revived again ..."

Q11. Ninf-G には共有メモリは用意されているか。

Q. Ninf-G を利用しようと考えているが、Ninf-G には何らかの共有メモリ機構は用意されているか? Ninf-G Executable 間で変数の値を共有することはできないか?

A. Ninf-G には、共有メモリ機能は無い。Ninf-G Executable 間で変数の値を共有する機構は用意されていない。

Q12. コンパイラやリンカの設定はできるか。

Q. コンパイラやリンカの設定はできるか?

A. 可能である。IDL ファイルの構文には、**Compiler, Linker** などが用意されている。

Q13. ジョブがすぐに実行されない。

Q. 関数ハンドルを作成しても、ジョブがすぐに実行されない。そのため、`grpc_call()` API や `grpc_call_async()` API が止まってしまう。このようなことは起こるのか？

A. 起こり得る。例えば、ジョブ投入先のクラスタの利用者が多く、他のジョブが優先されて実行される場合は、他のジョブが終了するまで待たされることがある。必ずすぐに実行されるわけではない。

もし、ジョブの起動時間にタイムリミットを設けたい場合は、クライアントコンフィギュレーションファイルの `<SERVER>` セクションに、`job_startTimeout` 属性を設定すればよい。

Q14. `workDirectory` とは何か。

Q. クライアントコンフィギュレーションファイルに設定可能な属性 `workDirectory` とは何か？

A. `workDirectory` とは、**Ninf-G Executable** のプロセスが動作するためのディレクトリを指定する機能である。デフォルトでは、**Ninf-G Executable** は、**Ninf-G Executable** プログラムが置かれているディレクトリにて動作する。このディレクトリを変更したい場合に利用する。

RPC 関数/計算関数にて、相対パス指定によりファイルをオープンした場合は、この `workDirectory` を起点としてファイルがオープンされる。

また、計算関数の不具合により **Segmentation fault** が起こった場合には、異常終了時のメモリイメージである `core` ファイルが出力される場合があるが、この `core` ファイルが出力されるディレクトリも `workDirectory` に指定されたディレクトリとなる。

指定された `workDirectory` が存在しない場合には、エラーとなるため注意が必要である。

クラスタのジョブ投入ノードと実際にジョブが実行される計算ノードが別である場合にも注意する必要がある。**Globus Toolkit** の制約により、例え計算ノードには `workDirectory` が存在していても、ジョブ投入ノードには指定された `workDirectory` が存在しなければエラーとなってしまう。

Q15. `stdout, stderr` がクライアントに転送されない。

Q. IDL ファイルや計算関数中に、`printf()` や `fprintf()` によって文字列を出力したが、その表示がクライアントに転送されない。クライアントコンフィギュレーションファイルには `<SERVER>` セクションに `redirect_outerr true` と設定している。なぜ転送されないのか？

A. **Ninf-G** は、計算関数内の `stdout, stderr` の転送のため、**Globus Toolkit GASS** モジュールの提供する、`stdout, stderr` 転送機能を利用している。

GASS stdout, stderr 転送機能では、転送タイミングが Globus Toolkit GRAM が利用する jobmanager 毎に異なる。そのため、転送タイミングは同じではない。

例えば、`jobmanager-fork` を利用すると、1 行出力される毎に数秒後にはクライアントの `stdout, stderr` に転送され表示される。

しかし、別の `jobmanager` の場合、ジョブの終了時に出力内容が一気に転送されることもある。この場合、Ninf-G の関数ハンドルを `grpc_function_handle_destruct()` によって破棄するまでは転送されないことになる。

また、なんらかの理由でジョブがキャンセルされると転送されないこともある。Ninf-G の場合、当該ハンドルとの間に接続の異常切断など何らかのエラーが発生すると安全のためジョブのキャンセルを発行する。

そして、Ninf-G クライアントが `grpc_function_handle_destruct()` を実行する前に `exit()` などにより終了してしまった場合も転送されない。

Q16. ローカル LDIF ファイルの作成方法を知りたい。

Q. 私の利用するクラスタシステムでは、関数ハンドル作成時の指定先ホストである GRAM ジョブ投入ノード(例: `server01.example.org`) にはログインが許可されておらず、`server01.example.org` での Ninf-G Executable のコンパイルができない。
そのため `server01.example.org` 用のローカル LDIF ファイルが作成されない。どのようにすればよいか?

A. ローカル LDIF ファイルはテキストファイルであり、内容はユーザが書き換え可能である。

例えば、以下のようにすれば `server01.example.org` 用のローカル LDIF ファイルが得られる。

- まず、対象となる RPC 用 IDL ファイルを `server01.example.org` 以外のホスト(例: `client.example.org`)でコンパイルする。すると `client.example.org` 用のローカル LDIF ファイルが作成される。
- 次に、そこで得られたローカル LDIF ファイルをテキストエディタにて書き換える。

書き換える部分は、ローカル LDIF ファイル中のホスト名が記述されている部分のすべてである。

(`client.example.org` を `server01.example.org` に書き換える。)

- ローカル LDIF ファイルのファイル名にも、ホスト名が含まれているが、こちらはクライアントの実行には影響がない。ただし、混乱の原因となり得るため、ファイル名も `server01.example.org` へと変更した方がよい。

Ninf-G クライアントコンフィギュレーションファイルでは、上記の手順で作成したローカル **LDIF** ファイルを指定するとよい。

尚、ローカル **LDIF** ファイルのユーザによる書き換えについては、自己責任で行うこと。

以上.