GridRPCを用いた タスクファーミングAPIの試作

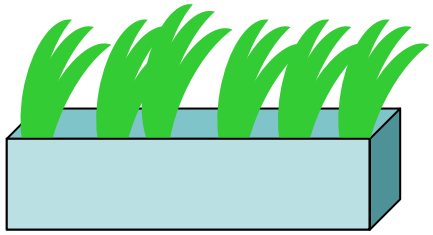
中田 秀基　(産総研、東工大)
田中 良夫　(産総研)
松岡 聡　(東工大、国情研)
関口 智嗣　(産総研)

Grid Technology Research Center AIST

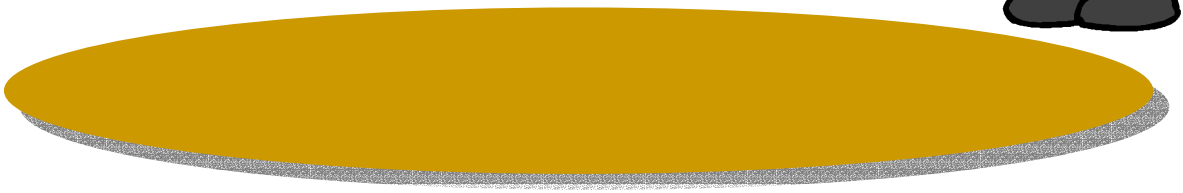
AIST

背景：タスクファーミング

- タスクファーミングとは
  - 大量の同じようなタスクを、大量の計算機上で実行
  - 多くの計算がタスクファーミングで実現できる
    - パラメータ・サーベイ
    - マスタ・ワーカ



タスクプール



プロセッサファーム

背景: タスクファーミング (2)

- GridRPC API では不十分
  - 可能ではあるが、たやすくはない
  - 耐故障性、スケジューリングがAPI仕様で保障されていないため、この部分を独自実装しなければならない
    - 実装によっては提供しているものもあるがAPI仕様では保障していない。

目的

- タスクファーミング用のAPIを定義し、GridRPC API上に実装する
  - ユーザにより使いやすいAPIを提供
  - 耐故障性とスケジューリング機能を実装
  - GridRPC API の有用性を確認、不足がないかをチェック

発表のアウトライン

- GridRPC API の概要
- タスクファーミングAPI
  - 要請
  - 設計
  - サンプルプログラム
  - 実装
- 議論
- まとめと今後の課題

GridRPC とは

- Grid 上で Remote Procedure Call を行うシステム
  - クライアント・サーバ型計算
  - サーバ側の関数を、クライアントに存在するかのように呼び出す機構を提供

```
double A[n][n],B[n][n],C[n][n];      /* Data Decl.*/
dmmul(n,A,B,C);                      /* Call local function*/
```

```
Ninf_call("dmmul",n,A,B,C); /*  Call server side routine*/
```

- GGFのGridRPC-WGで標準化が進行中

GridRPC API (1) 基本ストラクチャ

- 関数ハンドル
  - サーバ側の実行ファイルを抽象化
  - ハンドルを生成しておいて、ハンドルに対して呼び出しを行う
  - grpc_function_handle_t として実装
- セッションID
  - 個々の呼び出しを識別するID
  - int として実装
- 引数スタック
  - 動的に引数列をプログラムで生成する場合に使用
  - grpc_arg_stack

GridRPC API (2) 初期化と後処理

- `int grpc_initialize( char * config_file_name);`
  - コンフィギュレーションファイルを読み出して、RPCシステムに必要な初期化を行う。
- `int grpc_finalize();`
  - RPCに使用した資源を解放する。

GridRPC API (3) リモート関数ハンドル

```
int grpc_function_handle_default(
  grpc_function_handle_t * handle,
  char                   * func_name);
```

- デフォルトのホスト、ポートを使用して、第1引数で与えられる構造体領域を初期化

```
int grpc_function_handle_init(
   grpc_function_handle_t * handle,
   char * host_name, int port,
   char * func_name);
```

- サーバのホストとポートを明示的に指定して初期化

GridRPC API (4) RPC呼び出し

- `int grpc_call( grpc_function_handle_t *, ...);`
  - 第1引数で指定したハンドラを使用してブロッキングでRPC呼び出しを行う
- `int grpc_call_async( grpc_function_handle_t *, ...);`
  - 第1引数で指定したハンドラを使用してノンブロッキングでRPC呼び出しを行う

GridRPC API (5)　RPC呼び出し スタック版

- ```
  int grpc_call_arg_stack(
      grpc_function_handle_t * handle,
      grpc_arg_stack         * args);
  ```
  - 第1引数で指定したハンドラを使用してブロッキングでRPC呼び出しを行う
- ```
  int grpc_call_arg_stack_async(
      grpc_function_handle_t * handle,
      grpc_arg_stack         * args);
  ```
  - 第1引数で指定したハンドラを使用してノンブロッキングでRPC呼び出しを行う

GridRPC API (6) RPCの状態チェック

- `int grpc_probe(int sessionID);`
  - 第1引数で指定する呼び出しが終了したかどうかを調査
- `int grpc_cancel(int sessionID);`
  - 実行中の関数をキャンセル

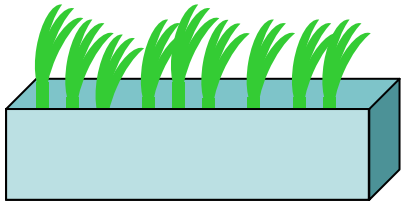
- `int grpc_wait(int sessionID);`
  - 第1引数で指定する呼び出しの終了を待つ

GridRPC API (7) RPC Wait関数

- `int grpc_wait_and( int * idArray, int length);`
  - 複数の呼び出しがすべて終了するのを待つ。
- `int grpc_wait_or( int * idArray, int length, int * idPtr);`
  - 複数の呼び出しのいずれかが終了するのを待つ。
- `int grpc_wait_all();`
  - それまでに行ったノンブロッキング呼び出しがすべて終了するのを待つ。
- `int grpc_wait_any( int * idPtr);`
  - それまでに行ったノンブロッキング呼び出しのいずれかが終了するのを待つ。

GridRPC API によるタスクファーミング(1)

- 直感的な手法はうまくいかない
  - ラウンドロビンでサーバを決定し単純にRPCを繰り返す



サーバ群

GridRPC API によるタスクファーミング(2)

- 原因
  - シングルCPUのサーバに同時に複数のタスクを割り当てても意味がない
    - バッチシステムの場合
      - キューにたまるだけ→スケジューリングの妨げ
    - フォークする場合
      - 複数のタスクが同時に走り効率が低下

➔スロットリング

  - タスクの粒度がそろっていない場合や、サーバ性能がヘテロな場合は、負荷が不均等に

➔スケジューリング

GridRPC API によるタスクファーミング (3)

- grpc_wait_orを用いてスロットリングとスケジューリング
  - 不可能ではないが煩雑ー　ユーザの負担大

```
/* 関数ハンドルの初期化 */
 for (i = 0; i < NUM_HOSTS; i++)
   grpc_function_handle_init(&handles[i], hosts[i], port, "pi/pi_trial");
 /* サーバの数だけタスクを投入 */
 for (i = 0; i < NUM_HOSTS; i++)
   ids[i] =  grpc_call_async(&handles[i], i,  times, &count[i]);
 /* サーバを使いまわしてスケジューリング */
 while (1) {
   int i = 0, id, code;
   code = grpc_wait_any(&id);
   if (code == GRPC_OK && id == GRPC_OK)  break;      /* 終了 */
   if (code == GRPC_ERROR)   continue;

   for (i = 0; i < NUM_HOSTS; i++)  /* FIND HOST */
     if (ids[i] == id) break;
   sum += count[i];   done += times;
   if (done >= whole_times)
     continue;
   ids[i] = grpc_call_async(&handles[i], i, times, &count[i]);
 }
```

タスクファーミングライブラリへの要請

- 耐故障性
  - 失敗したタスクの自動再実行
- スケジューリングとスロットリング
  - 自動的に空きサーバを選択
  - 空きサーバができるまでは実行を行わない
- ファーミング中のユーザ処理
  - 実行中にユーザプログラムからメモリ管理などの処理を実行

タスクファーミングライブラリへの要請 (2)

- 仮定
  - 100 サーバ
  - 10000 タスク
  - 1タスクあたり、1Mbyteの入力と出力
  - 10000 x 2M = 20Gbyte のメモリが必要
- 同時にメモリ空間内に収めることは不可能
  - うまく実行を制御し、実行中にメモリの管理を行うことができればフットプリントは

    100 x 2M = 200M byte

タスクファーミングAPIの設計

- コールバック関数を使用
- プログラマが2つのコールバック関数を定義
  - Cleanup 関数
    - サーバに送るデータが使用していたメモリ領域を開放するための関数
  - Consume 関数
    - サーバから結果を受け取るために使用したメモリ領域を開放するための関数

API (1) - ng_group_t

- ng_group_t ー サーバプールを抽象化
- 初期化

```
int ng_group_init(
    ng_group_t          * group,
    char                * grpc_config_file,
    char                * machine_file,
    char                * function_name,
    ng_group_arg_cleanup_f    arg_cleanup,
    ng_group_result_consume_f result_consume
);
```

- 後処理

```
int ng_group_fin(
    ng_group_t              * group
);
```

API (1.5) プロパティ

プロパティの設定

```
int ng_group_set_property(
    ng_group_t          * group,
    int                 key,
    void                * val
);
```

API (2) - コールバック関数

- Cleanup ー送信引数領域を開放する関数
  - 引数スタック
  - シリアル番号
  - ユーザ定義データ

```
typedef void (* ng_group_arg_cleanup_f) (
  grpc_arg_stack * args,
  int            serial,
  void           * user_data
);
```

- Consume ー 結果領域を開放する関数
  - 引数スタック
  - シリアル番号
  - ユーザ定義データ
  - セッションID

```
typedef void (* ng_group_result_consume_f) (
  grpc_arg_stack * args,
  int            serial,
  void           * user_data,
  int            session_id
);
```

Grid Technology Research Center AIST

API (3) RPC 呼び出し

- 可変引数型と引数スタック型
  - すべてが非同期なので async はなし
- 空きサーバがなければブロック
  - 空きサーバが生じるのを待って自動的に復帰

```
int ng_group_call(
    ng_group_t          * group,
    void                * user_data,
    ...
);
```

```
int ng_group_call_arg_stack(
    ng_group_t          * group,
    void                * user_data,
    grpc_arg_stack      * args
);
```

API (4) 終了待ち関数

- ファーミングの終了を待つ
  - すべてのRPCとそのRPCに対応するConsume関数の終了を待つ

```
int ng_group_wait_done(
    ng_group_t              * group
);
```

プログラム例

```
long trial, in, count;

/* consume function */
void consume(grpc_arg_stack * stack, int  serial,
                    void * user_data,  int session_id){
  in += *((long *)user_data);
  free(user_data);
}
```

```
....

/* init the group */
 ng_group_init(&group, argv[1], argv[2],  "pi/pi_trial",
              NULL, consume) ;

 /* MAIN LOOP */
 while (no_of_trials > 0){
   no_of_trials -= count;
   long * out = (long *) malloc(sizeof(long));

   ng_group_call(&group, out, random(), count, out);
   trial += count;
 }

 ng_group_wait_done(&group);
 fprintf(stdout, "PI =:= %f¥n", 4.0 * (in / (double)(trial)));
 ng_group_fin(&group);
}
```

実装

- Ninf-G上に2つのバージョンを実装
  - スレッド版
    - Pthread を使用 - globus のpthread flavorを使用したNinf-Gと使用
  - 非スレッド版
    - non-thread flavor のNinf-Gで使用

ノンスレッド版実装

- ng_group_t 内にハンドルのプールを保持
  - 初期化時にすべてのサーバに対するハンドルを作成
- grpc_call_async と grpc_wait_or で実装
  - ng_group_call時に空きサーバがなければgrpc_wait_or
    でいずれかのセッションが終了し、空きサーバが生じ
    るのを待つ

スレッド版実装

- ng_group_t に呼び出しコンテクストをキューイング
  - 呼び出しコンテクスト - 引数列 と ユーザ定義データ
  - キューをバウンデッドバッファとして実装することでスロットリング
- サーバごとにスレッドを割り当て
  - 各スレッドがキューから呼び出しコンテクストを取り出して呼び出しを実行
  - grpc_wait_orを用いない

議論：NetSolve request farming API (1)

- NetSolve[UTK]のリクエストファーミング
  - イテレータオブジェクトを配列から生成して呼び出し

```
int size_array[200];
void *ptr_array[200];
void *sorted_array[200];

/* 配列のセットアップ */
size_array[0] = size1;
ptr_array[0] = ptr1;
sorted_array[0] = sorted1;
...
/* request farming の呼び出し */
status_array =  netsl_farm("i=0,199","iqsort",
                                ns_int_array(size_array,"$i"),
                                ns_ptr_array(ptr_array,"$i"),
                                ns_ptr_array(sorted_array,"$i"));
```

議論：NetSolve request farming API (2)

- 利点
  - 簡潔でわかりやすい
- 欠点
  - イテレータオブジェクトの文字列引数の意味が不明確
  - ファーミング中はプログラマはまったく制御ができない
    - すべてのデータをセットアップしてから呼び出す
    - 結果はすべてが終了してから処理
  - 結果に応じた新たなタスクの投入ができない
    - 利用できる問題のクラスが限定される

議論：Condor - MW

- Condor上に実装されたMWライブラリ
  - C++のクラステンプレートを提供
  - タスククラスとマスタ・ワーカクラスを実装
  - さまざまな通信路を実装
    - Socket, file, pvm
- タスクのマーシャリングをユーザが指定
- 初期化時以外にはタスクを生成することができない
  - 利用できる問題のクラスが限定される

議論: GridRPC API の問題点

- GridRPC API だけでは実装できないことが判明
  - 本実装はNinf-Gに依存
- grpc_arg_stack を va_list から生成する機能

```
int
grpc_arg_stack_create_from_va_list(
  grpc_function_handle_t * handle,
  va_list                * parms
  grpc_arg_stack         * stack
);
```

- grpc_arg_stack のコピーを生成する機能

```
int
grpc_arg_stack_copy(
  grpc_arg_stack * src,
  grpc_arg_stack * dst
);
```

- GGFのGridRPC-WGに提案

まとめ

- タスクファーミングＡＰＩの提案
  - 耐故障性とスケジューリングを提供
- 二通りの実装
  - スレッド版
  - ノンスレッド版
- サンプルプログラム

今後の課題

- タスクファーミングAPIの妥当性の評価
  - 大規模実アプリケーションの実装
- サーバ計算機の追加と削除
  - サーバ計算機を実行中にプログラム外部から追加削除できるインターフェイスを設ける
- ログ機能とログの視覚化
  - ファーミング進行状況を監視するために、ライブラリにログ出力機能を追加する
  - 出力されたログを視覚化し、解析を補助する外部プログラムを作成する