DMI ドキュメント

原 健太朗

2011.12.9

1　DMI プログラミングの基本

■ 1-1　プログラム

DMI では，実行途中でノード数が動的に増減するような再構成可能な並列プログラムを記述できるが，再構成についてはまだ実験段階にある機能であるため，ここでは再構成をともなわない並列プログラムの記述方法と実行方法を説明する．

プログラムの概形は以下のようになる：

```
#include "dmi_api.h" /* ヘッダファイル */

void DMI_main(int argc, char **argv) /* main() 関数 */
{
  ...;
  DMI_rescale(addr, node_num, thread_num);
  ...;
  return;
}

int64_t DMI_scaleunit(int my_rank, int pnum, int64_t addr) /* 各 DMI スレッドが実行する関
数 */
{
  ...;
  return ...;
}
```

この DMI プログラムを実行すると，1 個の DMI スレッドが生成されて，DMI_main() 関数を実行し始める．やがてこの DMI スレッドが，DMI_rescale(addr, node_num, thread_num) 関数を呼び出すと，計算に参加しているプロセス数が node_num 個になるまで待機する．プロセス数が node_num 個以上になった時点で，node_num 個の各プロセス上に thread_num 個のスレッドが生成される．つまり，合計 node_num×thread_num 個のスレッドが生成される．そして，これらの各スレッドは DMI_scaleunit(int my_rank, int pnum, int64_t addr) 関数を呼び出す．この

とき，pnum にはスレッド数が，my_rank にはそのスレッドのランク（0 以上 pnum 未満の一意な整数）が，addr には DMI_rescale() 関数に渡した addr がそのまま渡される．addr は 64 ビット整数である．したがって，DMI_scaleunit() 関数の中身は，my_rank と pnum を基にして SPMD 的に書けば良い．また，addr をグローバルアドレス空間のアドレスにして，そのグローバルアドレス空間にさまざまなデータを格納しておけば，DMI_main() 関数から DMI_scaleunit() 関数へ，任意のデータを渡すことができる．

詳細は，1-4 節で述べるサンプルプログラムを眺める方が早いと思うが，たとえば，横ブロック分割による行列行列積 $AB = C$ を計算する DMI プログラムは次のように書ける：

```
#include "dmi_api.h"

typedef struct scaleunit_t
{
  int32_t n; /* 行列のサイズ */
  int64_t a_addr; /* 行列 A */
  int64_t b_addr; /* 行列 B */
  int64_t c_addr; /* 行列 C */
  int64_t barrier_addr; /* バリア */
}scaleunit_t;

double sumof_matrix(double *matrix, int32_t n);

void DMI_main(int argc, char **argv)
{
  scaleunit_t scaleunit;
  int32_t n, init_node_num, thread_num, pnum;
  int64_t a_addr, b_addr, c_addr, barrier_addr, scaleunit_addr;

  if(argc != 4) {
    outn("usage : %s init_node_num thread_num n", argv[0]);
    error();
  }

  init_node_num = atoi(argv[1]); /* プロセス数 */
  thread_num = atoi(argv[2]); /* 1 プロセスあたりのスレッド数 */
  n = atoi(argv[3]); /* 行列のサイズ */
  pnum = init_node_num * thread_num; /* 全スレッド数 */

  catch(DMI_mmap(&scaleunit_addr, sizeof(scaleunit_t), 1, NULL)); /* 各スレッドへ渡す引数のためのグローバルアドレス空間確保 */
  catch(DMI_mmap(&barrier_addr, sizeof(DMI_barrier_t), 1, NULL)); /* バリアのためのグローバルアドレス空間確保 */
  catch(DMI_mmap(&a_addr, n / pnum * n * sizeof(double), pnum, NULL)); /* 行列 A のためのグローバルアドレス空間確保 */
  catch(DMI_mmap(&b_addr, n * n * sizeof(double), 1, NULL)); /* 行列 B のためのグローバルアドレス空間確保 */
  catch(DMI_mmap(&c_addr, n / pnum * n * sizeof(double), pnum, NULL)); /* 行列 C のためのグローバルアドレス空間確保 */
  catch(DMI_barrier_init(barrier_addr)); /* バリア初期化 */

  scaleunit.n = n;
```

```
  scaleunit.a_addr = a_addr;
  scaleunit.b_addr = b_addr;
  scaleunit.c_addr = c_addr;
  scaleunit.barrier_addr = barrier_addr;
  catch(DMI_write(scaleunit_addr, sizeof(scaleunit_t), &scaleunit, DMI_EXCLUSIVE, NULL));
       /* 引数をグローバルアドレス空間に書き込む */

  catch(DMI_rescale(scaleunit_addr, init_node_num, thread_num)); /* rescale */

  catch(DMI_barrier_destroy(barrier_addr)); /* バリア破棄 */
  catch(DMI_munmap(c_addr, NULL));
  catch(DMI_munmap(b_addr, NULL));
  catch(DMI_munmap(a_addr, NULL));
  catch(DMI_munmap(barrier_addr, NULL));
  catch(DMI_munmap(scaleunit_addr, NULL));
  return;
}

int32_t DMI_scaleunit(int my_rank, int pnum, int64_t scaleunit_addr) /* 各スレッド */
{
  int32_t i, j, k, n, nn;
  int64_t a_addr, b_addr, c_addr, barrier_addr;
  scaleunit_t scaleunit;
  double sum;
  double *original_a, *original_b, *original_c, *local_a, *local_b, *local_c;
  DMI_local_barrier_t *local_barrier;

  catch(DMI_read(scaleunit_addr, sizeof(scaleunit_t), &scaleunit, DMI_GET, NULL));
      /* 引数をグローバルアドレス空間から読み込む */
  bind_to_cpu(my_rank % PROCNUM); /* CPU を明示的に割りつける．(DMI にかぎらず MPI でも) 性能上とても
重要． */

  n = scaleunit.n;
  a_addr = scaleunit.a_addr;
  b_addr = scaleunit.b_addr;
  c_addr = scaleunit.c_addr;
  barrier_addr = scaleunit.barrier_addr;
  local_barrier = (DMI_local_barrier_t*)malloc(sizeof(DMI_local_barrier_t));
  catch(DMI_local_barrier_init(local_barrier, barrier_addr));
  nn = n / pnum;
  local_a = (double*)malloc(nn * n * sizeof(double)); /* 部分行列 A_i */
  local_b = (double*)malloc(n * n * sizeof(double)); /* 行列 B */
  local_c = (double*)malloc(nn * n * sizeof(double)); /* 部分行列 C_i */

  for(i = 0; i < nn; i++)
    for(j = 0; j < n; j++)
      local_c[i * n + j] = 0;

  mrand_init(516);
  original_a = NULL;
  original_b = NULL;
  original_c = NULL;
  if(my_rank == 0) {
    original_a = (double*)malloc(n * n * sizeof(double));
    original_b = (double*)malloc(n * n * sizeof(double));
```

```
    original_c = (double*)malloc(n * n * sizeof(double));

    for(i = 0; i < n; i++) {
      for(j = 0; j < n; j++) {
        original_a[i * n + j] = mrand_01(); /* 行列 A を初期化 */
        original_b[i * n + j] = mrand_01(); /* 行列 B を初期化 */
        original_c[i * n + j] = 0; /* 行列 C を初期化 */
      }
    }
  }

  catch(DMI_local_barrier_sync(local_barrier, pnum)); /* バリア */
  if(my_rank == 0) {
    catch(DMI_write(a_addr, n * n * sizeof(double), original_a, DMI_EXCLUSIVE, NULL));
    catch(DMI_write(b_addr, n * n * sizeof(double), original_b, DMI_EXCLUSIVE, NULL));
  }
  catch(DMI_local_barrier_sync(local_barrier, pnum)); /* バリア */
  catch(DMI_read(a_addr + my_rank * nn * n * sizeof(double), nn * n * sizeof(double),
        local_a, DMI_INVALIDATE, NULL)); /* 行列 A を Scatter */
  catch(DMI_read(b_addr, n * n * sizeof(double), local_b, DMI_INVALIDATE, NULL));
      /* 行列 B を Broadcast */

  for(i = 0; i < nn; i++) /* C_i = A_i x B */
    for(k = 0; k < n; k++)
      for(j = 0; j < n; j++)
        local_c[i * n + j] += local_a[i * n + k] * local_b[k * n + j];

  catch(DMI_write(c_addr + my_rank * nn * n * sizeof(double), nn * n * sizeof(double),
        local_c, DMI_EXCLUSIVE, NULL));
  catch(DMI_local_barrier_sync(local_barrier, pnum));
  if(my_rank == 0) {
    catch(DMI_read(c_addr, n * n * sizeof(double), original_c, DMI_INVALIDATE, NULL));
        /* 行列 C を Gather */

    sum = sumof_matrix(original_c, n);
    outn("# sum : %lf", sum);
    free(original_c);
    free(original_b);
    free(original_a);
  }

  catch(DMI_local_barrier_destroy(local_barrier));
  free(local_a);
  free(local_b);
  free(local_c);
  free(local_barrier);
  return 0;
}

double sumof_matrix(double *matrix, int32_t n)
{
  double sum;
  int32_t i, j;

  sum = 0;
```

```
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      sum += matrix[i * n + j];
  return sum;
}
```

■ 1-2 実行方法

詳細は dmimw -h を参照されたいが，もっとも基本的な実行方法は次のとおりである．

第一に，DMI プログラムをコンパイルする：

```
$ dmicc ex.c
```

なお，dmicc コマンドは内部的に gcc を呼び出しているので，gcc で利用できるコンパイルオプションはそのまま指定できる．

第二に，使用するノードを記述したファイルを作る．たとえばファイル名を node.txt とする：

```
hongo100
hongo101
hongo102
hongo103
hongo104
hongo105
hongo106
hongo107
hongo108
hongo109
```

第三に，DMI プログラムを実行するには以下のコマンドを打つ：

```
$ dmimw -f node.txt -n 8 ./a.out ...
```

すると，node.txt に記述されている先頭 8 個のノードの各ノードに，1 個ずつ DMI プロセスが生成される．つまり，hongo100，hongo101，hongo102，hongo103，hongo104，hongo105，hongo106，hongo107 の各ノードに 1 個ずつ，DMI プロセスが生成される．とくに，hongo100 では，1 個の DMI スレッドが生成されて DMI_main() 関数が実行され始める．この DMI スレッドはやがて DMI_rescale(addr, node_num, thread_num) 関数を呼び出すが，このとき node_num 個の DMI プロセスが生成されていないとブロックしてしまう．よって，-n オプションで指定するノード数は，node_num 個以上になるようにしておかなければならない．通常は，-n オプションで指定するノード数と，node_num の値を一致させて実行する．

たとえば，前述の行列行列積を 8 ノード ×4 スレッドで実行するのであれば，

```
$ dmimw -f node.txt -n 8 ./a.out 8 4 4096
```

として実行する．ちなみに，-n オプションの後にも「8 ノード」と指定し，./a.out の第 1 引数（や

がては DMI_rescale() 関数の第 1 引数として利用される）にも「8 ノード」と指定するのは一見無駄なように思え，実際に，再構成が関係しないプログラムでは無駄である．このような仕様は，再構成可能なプログラムを実行する場合に意味を帯びてくるのだが，ここでは触れない．

■ 1-3　インストール方法

以下の手順でインストールできる：

```
$ tar xvf dmi-X.X.X.X.tar.gz
$ cd dmi-X.X.X.X
$ ./configure -prefix=/home/xxxxx/install_directory
$ make
$ make install
```

「dmi-X.X.X.X」には実際のバージョン番号を入れる．なお，Subversion のリポジトリから取得した場合には，「dmi-X.X.X.X」の部分を「DMI」に読み替える．

■ 1-4　サンプルプログラム

sample ディレクトリの下には，以下のサンプルプログラムを入れてある．アルゴリズムに関しては論文 [1] の第 6 章を参照されたい：

ep_dmi.c　NAS Parallel Benchmark の EP.
mandel_dmi.c　マンデルブロ集合の描画.
matmat_dmi.c　横ブロック分割による行列行列積.
fox_dmi.c　Fox アルゴリズムによる行列行列積.
ssort_dmi.c　サンプリングソート.
nbody_dmi.c　N 体問題.
sor_dmi.c　ヤコビ法による PDE ソルバ.
fem_dmi.c　有限要素法による応力解析.
pagerank_dmi.c　ページランク計算.
minpath_dmi.c　最短路計算.
mutex_dmi.c　mutex の使用例.
allreduce_dmi.c　バリアや Allreduce の使用例.

これらのサンプルプログラムをコンパイルするには以下のコマンドを打つ：

```
$ cd dmi-X.X.X.X/sample
$ make
```

たとえば，各サンプルプログラムは以下のような引数で実行するとよい（fem_dmi.c, pagerank_dmi.c, minpath_dmi.c は，実行するために大規模なデータセットが必要なので省略する）．node.txt には 16

ノードが書かれているとし，各ノードには 8 スレッドを生成するものとする：

```
$ dmimw -f node.txt -n 16 ./ep_dmi 16 8 32
$ dmimw -f node.txt -n 16 ./mandel_dmi 16 8 1024 1024 1024 100000 0
$ dmimw -f node.txt -n 16 ./matmat_dmi 16 8 4096
$ dmimw -f node.txt -n 16 ./fox_dmi 16 8 121 6600
$ dmimw -f node.txt -n 16 ./nbody_dmi 16 8 24 10
$ dmimw -f node.txt -n 16 ./sor_dmi 16 8 512 10
$ dmimw -f node.txt -n 16 ./mutex_dmi 16 8 200
$ dmimw -f node.txt -n 16 ./allreduce_dmi 16 8 10
```

2 API 一覧

■ 2-1 基本事項

すべての API は DMI_xxxxx() という名前を持つ．また，すべての API は TRUE もしくは FALSE を返す．よって，エラーをチェックするには，

```
if(DMI_xxxxx(...) == FALSE)
  print_error();
```

というように，DMI の API を呼び出すたびにエラーをチェックしなければならないし，バグの箇所を特定するうえでは，このように各 API ごとにエラーをチェックすることはきわめて重要である．しかし，いちいち上記のように記述するのは面倒なので，catch() というマクロを利用すると便利である：

```
catch(DMI_xxxxx(...));
```

このように書いておくと，エラーが起きたときそれをわかりやすく表示してくれる．

■ 2-2 グローバルアドレス空間の確保/解放

- int32_t DMI_mmap(int64_t *dmi_addr_ptr, int64_t page_size, int64_t page_num, DMI_local_status_t *status)
  ページサイズが page_size のページを page_num 個持つグローバルアドレス空間を確保する．つまり，合計 page_size×page_num バイトのグローバルアドレス空間を確保する．確保したグローバルアドレス空間の先頭アドレスが dmi_addr_ptr に格納される．
- int32_t DMI_munmap(int64_t dmi_addr, DMI_local_status_t *status)
  先頭アドレスが dmi_addr であるようなグローバルアドレス空間を解放する．この dmi_addr は，いずれかの時点で DMI_mmap() 関数によって返された値でなければならない．つまり，Linux とは異なり，mmap したメモリ領域の途中を munmap するようなことはできない．
- int32_t DMI_malloc(int64_t *addr_ptr, int64_t size, int64_t page_size)
- int32_t DMI_realloc(int64_t old_addr, int64_t *new_addr_ptr, int64_t

size, int64_t page_size)

- int32_t DMI_free(int64_t addr)
  libc が mmap() 関数と munmap() 関数の上位関数として malloc() 関数，realloc() 関数，free() 関数を提供しているのと同様に，DMI では，DMI_mmap() 関数と DMI_munmap() 関数の上位関数として，DMI_malloc() 関数，DMI_realloc() 関数，DMI_free() 関数を提供している．ここでの性能上の問題は，DMI_malloc() 関数や DMI_realloc() 関数が内部的に DMI_mmap() 関数を呼び出すときにページサイズをいくつにとるかということであるが，それを page_size として指定できる．性能を気にする場合，これらの関数は使わない方がよい．

■ 2-3 read/write

- int32_t DMI_read(int64_t dmi_addr, int64_t size, void *in_ptr, int8_t mode, DMI_local_status_t *status)
  グローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータを，ローカルアドレス領域 [in_ptr, in_ptr+size) に読み込む．選択的キャッシュ read のモードとして，GET，INVALIDATE，UPDATE を mode に指定できる．
- int32_t DMI_write(int64_t dmi_addr, int64_t size, void *out_ptr, int8_t mode, DMI_local_status_t *status)
  ローカルアドレス領域 [out_ptr, out_ptr+size) のデータを，グローバルアドレス領域 [dmi_addr, dmi_addr+size) に書き込む．選択的キャッシュ write のモードとして，PUT，EXCLUSIVE を mode に指定できる．
- int32_t DMI_watch(int64_t dmi_addr, int64_t size, void *in_ptr, void *out_ptr, DMI_local_status_t *status)
  グローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータの変更を監視する．グローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータがローカルアドレス領域 [in_ptr, in_ptr+size) のデータと一致しているかぎり，DMI_watch() 関数はブロックする．グローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータがローカルアドレス領域 [in_ptr, in_ptr+size) のデータと一致しなくなった瞬間に，その時点におけるグローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータを，ローカルアドレス領域 [out_ptr, out_ptr+size) に書き込んで，DMI_watch() 関数が返る．機能的には，Linux における futex システムコールに近い．
- int32_t DMI_save(int64_t dmi_addr, int64_t size)
  DMI では，各 DMI プロセスにおけるメモリプールの使用量が一定値（デフォルトでは 32 GB）を超えた場合にページ置換が発動し，ページ置換アルゴリズムに基づいて適宜ページの追い出しが行われる．デフォルトではすべてのページが追い出し対象になるが，DMI_save() 関数を呼び出すことで，グローバルアドレス領域 [dmi_addr, dmi_addr+size) が属するページを追い出し対象から外すことができ，その DMI プロセスのメモリプールに確実に固定することができる．機能的には，Linux における mlock システムコールに近い．使用頻度は少ない．

- int32_t DMI_unsave(int64_t dmi_addr, int64_t size)
  グローバルアドレス領域 [dmi_addr, dmi_addr+size) が属するページを追い出し対象に含める．デフォルトではすべてのページが追い出し対象になっているため，DMI_unsave() 関数を呼び出す意味のある場面は，DMI_save() 関数によっていったん追い出し対象から外したページをふたたび追い出し対象に含めたくなった場合にかぎられる．機能的には，Linux における munlock システムコールに近い．使用頻度は少ない．

■ 2-4 read-modify-write

- int32_t DMI_atomic(int64_t dmi_addr, int64_t _size, void *_out_ptr, int64_t _out_size, void *_in_ptr, int64_t _in_size, int8_t _tag, int8_t mode, DMI_local_status_t *status)
  compare-and-swap や fetch-and-store などの read-modify-write 命令を自由に作り出すことができる．簡単にいえば，DMI_function() という名前の関数を定義して，その中に好きな read-modify-write 命令を好きな数だけ定義する．そのうえで，DMI_atomic() 関数を呼び出すと，DMI_function() 関数の中に定義された read-modify-write 命令が実行される．このときどの read-modify-write 命令が実行されるかは，tag という整数値で識別する．この API は強力である一方で使用方法が非常に煩雑なので，以下の説明は，本節の最後に述べる実際の使用方法とあわせて参照されたい．論文 [1] の第 3.5.2 項も参照されたい．
  まず，DMI プログラム中に DMI_function(void *page_ptr, int64_t size, void *out_ptr, int64_t out_size, void *in_ptr, int64_t in_size, int8_t tag) という関数を定義し，この関数内に任意の read-modify-write を記述しておく．ここで，page_ptr はこの read-modify-write を適用する対象となるグローバルアドレス空間上のデータ，size はそのサイズ，out_ptr は read-modify-write への入力データ，out_size はそのサイズ，in_ptr は read-modify-write の出力データ，in_size はそのサイズ，tag はタグである．tag は read-modify-write の識別番号であり，DMI_function() 関数のなかには，tag の値に応じて複数の read-modify-write を記述することができる．次に，int32_t DMI_atomic(int64_t dmi_addr, int64_t _size, void *_out_ptr, int64_t _out_size, void *_in_ptr, int64_t _in_size, int8_t _tag, int8_t mode, DMI_local_status_t *status) という関数を呼び出す．ここで，[dmi_addr, dmi_addr+_size) は read-modify-write を適用するグローバルアドレス領域であり，1 個のページに収まっている必要がある．さて，DMI_function() 関数を定義したうえで DMI_atomic() 関数を呼び出すと，DMI_atomic() 関数で指定した入力データ [_out_ptr, _out_ptr+_out_size) が，DMI_function() 関数の [out_ptr, out_ptr+out_size) として渡される．また，DMI_atomic() 関数で指定した_tag の値が DMI_function() 関数の tag として渡され，DMI_atomic() 関数で指定した_size の値が DMI_function() 関数の size として渡され，DMI_atomic() 関数で指定した_in_size の値が DMI_function() 関数の in_size として渡され，DMI_atomic() 関数で指定した_out_size の値が DMI_function() 関数の out_size として渡される．さらに，DMI_function() 関数

の [page_ptr, page_ptr+size) には，グローバルアドレス dmi_addr から_size バイト分のデータ本体が渡される．そして，tag の値に応じて，DMI_function() 関数がグローバルアドレス空間上のデータ [page_ptr, page_ptr+size) に対して何らかの read-modify-write を実行したあと，出力データ [in_ptr, in_ptr+in_size) にデータを格納すると，それが DMI_atomic() 関数の引数 [_in_ptr, _in_ptr+_in_size) として返る．まとめると，DMI_function() 関数には，「[out_ptr, out_ptr+out_size) を入力データとして [in_ptr, in_ptr+in_size) を出力データとするような，グローバルアドレス空間上のデータ [page_ptr, page_ptr+size) に対する任意の read-modify-write」を各 tag ごとに記述しておき，それを実行するためには DMI_atomic() 関数を呼び出せばよい．なお，mode には，選択的キャッシュ write のモードとして，DMI_EXCLUSIVE または DMI_PUT を指定できる．

以降の DMI_cas() 関数，DMI_fas() 関数，DMI_fad() 関数は，後述するように，いずれも DMI_atomic() 関数を使うことで簡単に実現できるものである．ただし，使用頻度が多いので，あえて独立した API として提供している．

- int32_t DMI_cas(int64_t dmi_addr, int64_t size, void *cmp_ptr, void *swap_ptr, int8_t *cas_flag_ptr, int8_t mode, DMI_local_status_t *status)

  compare-and-swap を行う．「グローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータがローカルアドレス領域 [cmp_ptr, cmp_ptr+size) のデータと一致していれば，グローバルアドレス領域 [dmi_addr, dmi_addr+size) にローカルアドレス領域 [swap_ptr, swap_ptr+size) のデータを格納したうえで，cas_flag_ptr に TRUE を格納する．一方で，グローバルアドレス領域 [dmi_addr, dmi_addr+size) のデータがローカルアドレス領域 [cmp_ptr, cmp_ptr+size) のデータと一致していなければ，何も行わず，cas_flag_ptr に FALSE を格納する.」という操作をアトミックに実行する．mode には，選択的キャッシュ write のモードとして，DMI_EXCLUSIVE または DMI_PUT を指定できる．
- int32_t DMI_fas(int64_t dmi_addr, int64_t size, void *out_ptr, void *in_ptr, int8_t mode, DMI_local_status_t *status)

  fetch-and-store を行う．「その時点でグローバルアドレス領域 [dmi_addr, dmi_addr+size) に格納されているデータをローカルアドレス領域 [in_ptr, in_ptr+size) に格納したあとで，グローバルアドレス領域 [dmi_addr, dmi_addr+size) にローカルアドレス領域 [out_ptr, out_ptr+size) のデータを格納する.」という操作をアトミックに行う．mode には，選択的キャッシュ write のモードとして，DMI_EXCLUSIVE または DMI_PUT を指定できる．
- int32_t DMI_fad(int64_t dmi_addr, int64_t add_value, int64_t *fetch_value_ptr, int8_t mode, DMI_local_status_t *status)

  fetch-and-add を行う．「グローバルアドレス領域 [dmi_addr, dmi_addr+64) に格納されている 64 ビット整数の値をローカルアドレス領域 [fetch_value_ptr, fetch_value_ptr+64) に格納したあとで，グローバルアドレス領域 [dmi_addr, dmi_addr+64) に格納されている 64 ビット整数に add_value を加算する.」という操作をアトミックに実行する．グローバルアドレス dmi_addr はちょうど 64 ビットの大きさを持っている必要がある．mode には，選択的キャッシュ write のモー

ドとして，DMI_EXCLUSIVE または DMI_PUT を指定できる．

DMI_atomic() 関数の使用例を以下に示す．たとえば，DMI_cas() 関数と DMI_fas() 関数と DMI_fad() 関数は，DMI_atomic() 関数を使って次のように実装できる：

```
#define CAS_TAG 12345
#define FAS_TAG 12346
#define FAD_TAG 12347

void DMI_function(void *page_ptr, int64_t copy_size, void *out_ptr, int64_t out_size,
                  void *in_ptr, int64_t in_size, int8_t tag)
{
  int64_t uoffset, usize, size, object_offset;
  void *cmp_ptr, *swap_ptr, *out2_ptr;

  switch(tag) {
    case CAS_TAG:
      uoffset = 0;
      usize = copy_size;
      cmp_ptr = (int8_t*)out_ptr + uoffset;
      uoffset += usize;
      usize = copy_size;
      swap_ptr = (int8_t*)out_ptr + uoffset;
      uoffset += usize;

      if(memcmp(page_ptr, cmp_ptr, copy_size) == 0) {
        memcpy(page_ptr, swap_ptr, copy_size);
        if(in_size > 0) {
          ((int8_t*)in_ptr)[0] = TRUE;
        }
      } else {
        if(in_size > 0) {
          ((int8_t*)in_ptr)[0] = FALSE;
        }
      }
      break;
    case FAD_TAG:
      if(in_ptr != NULL) {
        *(int64_t*)in_ptr = *(int64_t*)page_ptr;
      }
      *(int64_t*)page_ptr += *(int64_t*)out_ptr;
      break;
    case FAS_TAG:
      if(in_size > 0) {
        memcpy(in_ptr, page_ptr, copy_size);
      }
      memcpy(page_ptr, out_ptr, copy_size);
      break;
    }
}

void DMI_cas_example(int64_t dmi_addr, int64_t size, void *cmp_ptr, void *swap_ptr,
                     int8_t *cas_flag_ptr, int8_t mode)
{
```

```
  int8_t *buf;
  int64_t uoffset, usize;

  buf = malloc(size + size);
  uoffset = 0;
  usize = size;
  memcpy(buf + uoffset, cmp_ptr, usize);
  uoffset += usize;
  usize = size;
  memcpy(buf + uoffset, swap_ptr, usize);
  uoffset += usize;
  catch(DMI_atomic(dmi_addr, size, buf, uoffset, cas_flag_ptr,
                   sizeof(int8_t), CAS_TAG, mode, NULL));
  return;
}

void DMI_fas_example(int64_t dmi_addr, int64_t size, void *out_ptr,
                     void *in_ptr, int8_t mode)
{
  catch(DMI_atomic(dmi_addr, size, out_ptr, size, in_ptr, size, FAS_TAG, mode, NULL));
  return;
}

void DMI_fad_example(int64_t dmi_addr, int64_t add_value,
                     int64_t *fetch_value_ptr, int8_t mode)
{
  catch(DMI_atomic(dmi_addr, sizeof(int64_t), &add_value, sizeof(int64_t),
                   fetch_value_ptr, sizeof(int64_t), FAD_TAG, mode, NULL));
  return;
}
```

■ 2-5 離散 read/write のグルーピング

DMI_read() 関数/DMI_write() 関数では，ある連続的なグローバルアドレス領域しか read/write することができないが，離散 read/write のグルーピングを利用すると，離散的なグローバルアドレス領域を一括して性能よく read/write することができる．いま一括して read/write したいグローバルアドレス領域を，$[a_0, a_0 + s_0)$，$[a_1, a_1 + s_1)$，…，$[a_{n-1}, a_{n-1} + s_{n-1})$ とする．また，これらのグローバルアドレス領域上のデータに対応するローカルアドレス領域を $[b + o_0, b + o_0 + s_0)$，$[b + o_1, b + o_1 + s_1)$，…，$[b + o_{n-1}, b + o_{n-1} + s_{n-1})$ とする．つまり，read の場合には，グローバルアドレス領域 $[a_i, a_i + s_i)$ のデータをローカルアドレス領域 $[b + o_i, b + o_i + s_i)$ に read したいとし，write の場合には，ローカルアドレス領域 $[b + o_i, b + o_i + s_i)$ のデータをグローバルアドレス領域 $[a_i, a_i + s_i)$ に write したいとする．ここで，a_0，a_1，…，a_{n-1} はグローバルアドレス，s_0，s_1，…，s_{n-1} はサイズ，b はローカルアドレス（の「ベースポインタ」），o_0，o_1，…，o_{n-1} は b からのオフセットである．

- int32_t DMI_group_init(DMI_local_group_t *group, int64_t *addrs, int64_t *ptr_offsets, int64_t *sizes, int32_t group_num)
  離散アクセスのグループ group を作成して初期化する．a_0，a_1，…，a_{n-1} を配列 addrs として渡

す．o_0，o_1，...，o_{n-1} を配列 ptr_offsets として渡す．s_0，s_1，...，s_{n-1} を配列 sizes として渡す．n を group_num として渡す．このように，グループの作成時に，a_i，o_i，s_i は決定する必要があるが，b は各 read/write ごとに変更することができる．

- int32_t DMI_group_destroy(DMI_local_group_t *group)
  離散アクセスのグループ group を破棄する．
- int32_t DMI_group_read(DMI_local_group_t *group, void *in_ptr, int8_t mode, DMI_local_status_t *status)
  この read における b を in_ptr として渡す．選択的キャッシュ read のモードとして，DMI_GET，DMI_INVALIDATE，DMI_UPDATE を mode に指定できる．ほとんどの場合，DMI_GET を指定することになる．なぜなら，DMI_INVALIDATE や DMI_UPDATE を指定するとページ単位の転送が起きてしまうため，離散的に read を行っている意味がなくなってしまうためである．
- int32_t DMI_group_write(DMI_local_group_t *group, void *out_ptr, int8_t mode, DMI_local_status_t *status)
  この write における b を out_ptr として渡す．選択的キャッシュ write のモードとして，DMI_PUT，DMI_EXCLUSIVE を mode に指定できる．ほとんどの場合，DMI_PUT を指定することになる．なぜなら，DMI_EXCLUSIVE を指定するとページ単位の転送が起きてしまうため，離散的に write を行っている意味がなくなってしまうためである．

使用例を以下に示す：

```
DMI_group_t read_group;
int64_t element_num, i;
int64_t *values, *addrs, *offsets, *sizes;

element_num = 1024;
values = (int64_t*)malloc(sizeof(int64_t) * element_num);
addrs = (int64_t*)malloc(sizeof(int64_t) * element_num);
offsets = (int64_t*)malloc(sizeof(int64_t) * element_num);
sizes = (int64_t*)malloc(sizeof(int64_t) * element_num);
for(i = 0; i < element_num; i++) {
  addrs[i] = wherever_you_want_to_access(i) * sizeof(int64_t);
  offsets[i] = i * sizeof(int64_t);
  sizes[i] = sizeof(int64_t);
}

catch(DMI_group_init(&read_group, addrs, offsets, sizes, element_num));
catch(DMI_group_read(&read_group, values, DMI_GET, NULL));
catch(DMI_group_destroy(&read_group));
free(values);
free(addrs);
free(offsets);
free(sizes);
```

■ 2-6 関数の非同期化

DMI_xxxxx(..., DMI_local_status_t *status) のように, 最終引数に status を指定できる関数は, その関数を非同期化することができ, 後から関数の完了を待機することができる. 非同期化しない場合には, status に NULL を指定する. 非同期化する場合には, status に DMI_local_status_t 型の変数を指定すると, その status が非同期化のハンドルになり, 関数は完了を待つことなくすぐに返る.

- int32_t DMI_check(DMI_local_status_t *status, int32_t *ret_ptr)
  ハンドル status で表される非同期化関数の実行が完了したかどうかを調べる. 完了していれば, DMI_check() 関数は TRUE を返し, 非同期化関数の返り値が ret_ptr に格納される. 完了していなければ, DMI_check() 関数は FALSE を返す. DMI_check() 関数はつねにすぐに返る.
- void DMI_wait(DMI_local_status_t *status, int32_t *ret_ptr)
  ハンドル status で表される非同期化関数が完了するまで待機する. DMI_wait() 関数を呼び出した時点で関数の実行が完了していれば, その非同期化関数の返り値が ret_ptr に格納される. 完了していなければ, 非同期化関数の実行が完了するまで待機したあと, その非同期化関数の返り値が ret_ptr に格納される.

使用例を以下に示す:

```
DMI_local_status_t statuses[128];

for(i = 0; i < 128; i++) {
  catch(DMI_read(..., &statuses[i]));
}
...; /* 何か他の処理 */
for(i = 0; i < 128; i++) {
  DMI_wait(&statuses[i], &ret);
  if(ret == FALSE)
    error();
}
```

■ 2-7 バリア

- int32_t DMI_barrier_init(int64_t dmi_barrier_addr)
  バリア変数を初期化する. dmi_barrier_addr はグローバルアドレスで, DMI_barrier_t のサイズを持っていなければならない.
- int32_t DMI_barrier_destroy(int64_t dmi_barrier_addr)
  dmi_barrier_addr で表されるバリア変数を破棄する.
- int32_t DMI_local_barrier_init(DMI_local_barrier_t *barrier, int64_t dmi_barrier_addr)
  dmi_barrier_addr で表されるバリア変数を使うだけではバリア操作は実現できない. バリア操作を行おうとする各 DMI スレッドは, このバリア変数をもとにして, その DMI スレ

ッドにとってローカルなバリア変数を作る必要がある．DMI_local_barrier_init() 関数は，dmi_barrier_addr で表されるバリア変数に対応するローカルバリア変数を初期化する．barrier は DMI_local_barrier_t 型の変数でなければならない．

- int32_t DMI_local_barrier_destroy(DMI_local_barrier_t *barrier)
  barrier で表されるローカルバリア変数を破棄する．
- int32_t DMI_local_barrier_sync(DMI_local_barrier_t *barrier, int32_t pnum)
  barrier で表されるローカルバリア変数を使って，pnum 個の DMI スレッドでバリア操作を行う．この DMI_local_barrier_sync() 関数は，合計 pnum 個の DMI スレッドが DMI_local_barrier_sync() 関数を呼び出した瞬間に返る．
- int32_t DMI_local_barrier_allreduce(DMI_local_barrier_t *barrier, int32_t pnum, void *sub_value_ptr, void *value_ptr, int8_t op_type, int8_t type_type)
  barrier で表されるローカルバリア変数を使って，pnum 個の DMI スレッドで Allreduce 操作を行う．この DMI_local_barrier_allreduce() 関数は，合計 pnum 個の DMI スレッドが DMI_local_barrier_allreduce() 関数を呼び出した瞬間に返る．sub_value_ptr には各 DMI スレッドからの入力値（たとえば各 DMI スレッドが計算した部分和）を指定すると，Allreduce 操作の結果として得られた値が value_ptr（たとえば総和）に格納される．また，Allreduce 操作の種類を op_type に指定する．具体的には，op_type には，DMI_OP_MAX（最大値を求める），DMI_OP_MIN（最小値を求める），DMI_OP_SUM（和を求める），DMI_OP_PROD（積を求める）を指定できる．さらに，sub_value_ptr および value_ptr の型を type_type に指定する．具体的には，type_type には，DMI_TYPE_CHAR，DMI_TYPE_SHORT，DMI_TYPE_INT，DMI_TYPE_LONG，DMI_TYPE_LONGLONG，DMI_TYPE_FLOAT，DMI_TYPE_DOUBLE を指定できる．

使用例を以下に示す：

```
#include "dmi_api.h"

typedef struct scaleunit_t
{
  int64_t dmi_barrier_addr;
}scaleunit_t;

void DMI_main(int argc, char **argv)
{
  scaleunit_t scaleunit;
  int32_t init_node_num, thread_num;
  int64_t dmi_barrier_addr, scaleunit_addr;

  if(argc != 3) {
      outn("usage : %s init_node_num thread_num", argv[0]);
      error();
  }
```

```
  init_node_num = atoi(argv[1]);
  thread_num = atoi(argv[2]);

  catch(DMI_mmap(&scaleunit_addr, sizeof(scaleunit_t), 1, NULL));
  catch(DMI_mmap(&dmi_barrier_addr, sizeof(DMI_barrier_t), 1, NULL)); /* バリア変数のための
グローバルアドレス空間確保 */
  catch(DMI_barrier_init(dmi_barrier_addr)); /* バリア変数の初期化 */
  scaleunit.dmi_barrier_addr = dmi_barrier_addr;
  catch(DMI_write(scaleunit_addr, sizeof(scaleunit_t), &scaleunit, DMI_EXCLUSIVE, NULL));

  catch(DMI_rescale(scaleunit_addr, init_node_num, thread_num)); /* rescale */

  catch(DMI_barrier_destroy(dmi_barrier_addr)); /* バリア変数の破棄 */
  catch(DMI_munmap(scaleunit_addr, NULL));
  catch(DMI_munmap(dmi_barrier_addr, NULL));
  return;
}

int32_t DMI_scaleunit(int my_rank, int pnum, int64_t scaleunit_addr)
{
  DMI_local_barrier_t barrier; /* ローカルバリア変数 */
  scaleunit_t scaleunit;
  int sum;

  catch(DMI_read(scaleunit_addr, sizeof(scaleunit_t), &scaleunit, DMI_GET, NULL));
  bind_to_cpu(my_rank % PROCNUM);
  catch(DMI_local_barrier_init(&barrier, scaleunit.dmi_barrier_addr)); /* ローカルバリア変
数の初期化 */
  catch(DMI_local_barrier_sync(&barrier, pnum)); /* バリア */
  catch(DMI_local_barrier_allreduce(&barrier, pnum, &my_rank, &sum,
        DMI_OP_SUM, DMI_TYPE_INT)); /* Allreduce */
  if(sum != pnum * (pnum - 1) / 2) error(); /* Allreduce の結果のチェック */
  catch(DMI_local_barrier_destroy(&barrier)); /* ローカルバリア変数の破棄 */
  return 0;
}
```

■ 2-8 排他制御変数

基本的に，pthread_mutex_xxxxx() と同様の API である．

- int32_t DMI_mutex_init(int64_t dmi_mutex_addr)
  排他制御変数を初期化する．dmi_mutex_addr はグローバルアドレスで，DMI_mutex_t のサイズを持っていなければならない．
- int32_t DMI_mutex_destroy(int64_t dmi_mutex_addr)
  dmi_mutex_addr で表される排他制御変数を破棄する．
- int32_t DMI_mutex_lock(int64_t dmi_mutex_addr)
  dmi_mutex_addr で表される排他制御変数を lock する．
- int32_t DMI_mutex_unlock(int64_t dmi_mutex_addr)
  dmi_mutex_addr で表される排他制御変数を unlock する．

- int32_t DMI_mutex_trylock(int64_t dmi_mutex_addr, int32_t *try_flag_ptr)
  dmi_mutex_addr で表される排他制御変数を trylock する．trylock した結果，lock が成功すれば try_flag_ptr に TRUE が，lock が失敗すれば try_flag_ptr に FALSE が格納される．

使用例を以下に示す：

```
#include "dmi_api.h"

typedef struct scaleunit_t
{
  int64_t mutex_addr;
  ...;
}scaleunit_t;

void DMI_main(int argc, char **argv)
{
  ...;
  catch(DMI_mmap(&mutex_addr, sizeof(DMI_mutex_t), 1, NULL)); /* mutex のためのグローバルア
ドレス空間確保 */
  catch(DMI_mutex_init(mutex_addr)); /* mutex を初期化 */
  ...;
  scaleunit.mutex_addr = mutex_addr;
  catch(DMI_write(scaleunit_addr, sizeof(scaleunit_t), &scaleunit, DMI_EXCLUSIVE, NULL));

  catch(DMI_rescale(scaleunit_addr, init_node_num, thread_num));

  catch(DMI_mutex_destroy(mutex_addr));
  catch(DMI_munmap(mutex_addr, NULL));
  return;
}

int32_t DMI_scaleunit(int my_rank, int pnum, int64_t scaleunit_addr)
{
  scaleunit_t scaleunit;

  catch(DMI_read(scaleunit_addr, sizeof(scaleunit_t), &scaleunit, DMI_GET, NULL));
  ...;
  catch(DMI_mutex_lock(scaleunit.mutex_addr)); /* mutex をロック */
  ...;
  catch(DMI_mutex_unlock(scaleunit.mutex_addr)); /* mutex をアンロック */
  ...;
  return 0;
}
```

■ 2-9　条件変数

基本的に，pthread_cond_xxxxx() と同様の API である．

- int32_t DMI_cond_init(int64_t dmi_cond_addr)
  条件変数を初期化する．dmi_cond_addr はグローバルアドレスで，DMI_cond_t のサイズを持っ

ていなければならない.

- int32_t DMI_cond_destroy(int64_t dmi_cond_addr)
  dmi_cond_addr で表される条件変数を破棄する.
- int32_t DMI_cond_wait(int64_t dmi_cond_addr, int64_t dmi_mutex_addr)
  dmi_cond_addr で表される条件変数の上で眠る.
- int32_t DMI_cond_signal(int64_t dmi_cond_addr)
  dmi_cond_addr で表される条件変数の上で眠っている DMI スレッドのうち，いずれか 1 個を起こす.
- int32_t DMI_cond_broadcast(int64_t dmi_cond_addr)
  dmi_cond_addr で表される条件変数の上で眠っている DMI スレッドをすべて起こす.

使用例は DMI_mutex_xxxxx() と同様である.

■ 2-10 スピンロック

基本的に，pthread_spinlock_xxxxx() と同様の API である.

- int32_t DMI_spin_init(int64_t dmi_spinlock_addr)
  スピンロックを初期化する．dmi_spinlock_addr はグローバルアドレスで，DMI_spinlock_t のサイズを持っていなければならない.
- int32_t DMI_spin_destroy(int64_t dmi_spinlock_addr)
  dmi_spinlock_addr で表されるスピンロックを破棄する.
- int32_t DMI_spin_lock(int64_t dmi_spinlock_addr)
  dmi_spinlock_addr で表されるスピンロックを lock する.
- int32_t DMI_spin_unlock(int64_t dmi_spinlock_addr)
  dmi_spinlock_addr で表されるスピンロックを unlock する.
- int32_t DMI_spin_trylock(int64_t dmi_spinlock_addr, int32_t *try_flag_ptr)
  dmi_spinlock_addr で表される排他制御変数を trylock する．trylock した結果，lock が成功すれば try_flag_ptr に TRUE が，lock が失敗すれば try_flag_ptr に FALSE が格納される.

使用例は DMI_mutex_xxxxx() と同様である.

■ 2-11 read-write-set

read-write-set の API については論文 [1] の第 5 章を参照されたい.

- int32_t DMI_rwset_init(int64_t dmi_rwset_addr, int64_t element_num, int64_t element_size, int32_t rwset_num)
  論文 [1] の第 5.2.2 項における rwset_init() 関数．read-write-set を初期化する．dmi_rwset_addr はグローバルアドレスで，DMI_rwset_t のサイズを持っていなければならない.

dmi_rwset_addr がこの read-write-set のハンドルになる．節点数を element_num に，領域数を rwset_num に指定する．

- int32_t DMI_rwset_destroy(int64_t dmi_rwset_addr)
  論文 [1] の第 5.2.2 項における rwset_destroy() 関数．read-write-set を破棄する．
- int32_t DMI_rwset_decompose(int64_t dmi_rwset_addr, int32_t my_id, int64_t *write_elements, int32_t write_element_num)
  論文 [1] の第 5.2.2 項における rwset_decompose() 関数．writeset を定義する．read-write-set のハンドルを dmi_rwset_addr に指定する．領域番号を my_id に指定する．領域 my_id の writeset を配列 write_elements に，領域 my_id の writeset の要素数を write_element_num に指定する．
- int32_t DMI_local_rwset_init(DMI_local_rwset_t *rwset, int64_t dmi_rwset_addr, int32_t my_id, int64_t *read_elements, int32_t read_element_num)
  論文 [1] の第 5.2.2 項における rwset_build() 関数．readset を定義する．read-write-set のハンドルを dmi_rwset_addr に指定する．領域番号を my_id に指定する．領域 my_id の readset を配列 read_elements に，領域 my_id の readset の要素数を read_element_num に指定する．
- int32_t DMI_local_rwset_destroy(DMI_local_rwset_t *rwset)
  DMI_local_rwset_init() 関数で確保したメモリを解放する．
- int32_t DMI_local_rwset_write(DMI_local_rwset_t *rwset, void *buf, DMI_local_status_t *status)
  論文 [1] の第 5.2.2 項における rwset_write() 関数．writeset 内の節点の値をローカルアドレス領域 buf に読み込む．
- int32_t DMI_local_rwset_read(DMI_local_rwset_t *rwset, void *buf, DMI_local_status_t *status)
  論文 [1] の第 5.2.2 項における rwset_read() 関数．ローカルアドレス領域 buf の値を readset の節点の値として書き込む．

■ 2-12 ノードの取り扱い

本節で述べる関数は，1-1 節で述べたプログラミングモデルを利用するだけであれば必要ない．

- int32_t DMI_rank(int32_t *dmi_id_ptr)
  この DMI プロセスの ID を dmi_id_ptr に格納する．なお，この ID は DMI プログラム全体のなかで一意である．
- int32_t DMI_nodes(DMI_node_t *dmi_node_array, int32_t *num_ptr, int32_t capacity)
  現在存在している DMI プロセスたちの情報を配列 dmi_node_array に格納し，格納した DMI プロセス数を num_ptr に格納する．配列 dmi_node_array に格納可能な DMI プロセス数を capacity で指定できる．つまり，実際に存在している DMI プロセス数が capacity より多け

れば，配列 dmi_node_array には capacity 個だけの DMI プロセスたちの情報が格納され，num_ptr には capacity の値が格納される．配列 dmi_node_array の各要素は DMI_node_t 型である必要がある．DMI_node_t 構造体は DMI プロセスの情報を表す以下のようなメンバを持っている：

dmi_id その DMI プロセスの ID.

core その DMI プロセスが属するノードの CPU 数.

memory その DMI プロセスが提供しているメモリプールの容量.

hostname その DMI プロセスのホスト名.

state その DMI プロセスの状態．参加を要求している状態であれば DMI_OPEN，実行中であれば DMI_OPENED，脱退を要求している状態であれば DMI_CLOSE，脱退後の状態であれば DMI_CLOSED になる．なお，DMI_nodes() 関数は現在存在している DMI プロセスたちの情報しか返さないので，DMI_CLOSED な状態を持つ DMI プロセスが返されることはない.

■ 2-13 スレッドの取り扱い

本節で述べる関数は，1-1 節で述べたプログラミングモデルを利用するだけであれば必要ない.

- int32_t DMI_create(DMI_thread_t *dmi_thread_ptr, int32_t dmi_id, int64_t dmi_addr, int64_t stack_size, DMI_local_status_t *status)
  dmi_id で表される ID を持つ DMI プロセス上に DMI スレッドを生成する．生成された DMI スレッドは DMI_thread(int64_t dmi_addr) 関数から実行を始めるが，このとき，DMI_create() 関数に渡した dmi_addr がそのまま DMI_thread() 関数に渡される．つまり，各 DMI スレッドに渡したいデータを格納したグローバルアドレス空間を用意しておいて，その先頭グローバルアドレスを dmi_addr に渡すことで，任意のデータを DMI スレッドへと渡すことができる．stack_size は，生成される DMI スレッドに与えるスタック領域のサイズである．生成された DMI スレッドのハンドルが dmi_thread_ptr に格納される．DMI_thread_t 構造体は，DMI スレッドの情報を表す以下の 2 つのメンバを持っている：

  dmi_id DMI スレッドが生成された DMI プロセスの ID.

  thread_id その DMI プロセスにおける DMI スレッドのスレッド ID．このスレッド ID は各 DMI プロセスのなかでは一意であるが，DMI プロセスが異なれば，同一のスレッド ID を持つ DMI スレッドが存在する可能性はある.

  status はこの関数を非同期化するためのハンドルである．関数の非同期化については 2-6 節で述べる．関数の非同期化を利用しない場合には，status に NULL を指定すればよい.
- int32_t DMI_join(DMI_thread_t dmi_thread, int64_t *dmi_addr_ptr, DMI_local_status_t *status)
  ハンドルが dmi_thread の DMI スレッドを回収する．DMI_thread() 関数の返り値が dmi_addr_ptr に格納される．返り値を得る必要がない場合には dmi_addr_ptr に NULL を指定する.

- int32_t DMI_detach(DMI_thread_t dmi_thread, DMI_local_status_t *status)
  ハンドルが dmi_thread の DMI スレッドをディタッチする.
- int32_t DMI_wake(DMI_thread_t dmi_thread, void *out_ptr, int64_t out_size, DMI_local_status_t *status)
  ハンドルが dmi_thread の DMI スレッドに対して起床通知を送る．起床通知と同時に，out_ptr からはじまる out_size バイトのデータを送ることができる．使用頻度は少ない.
- int32_t DMI_suspend(void *in_ptr)
  この DMI_suspend() 関数を呼び出した DMI スレッドを眠らせる．起床通知が届いた瞬間に，起床通知とともに送られて来たデータを in_ptr に格納して返る．DMI_suspend() 関数は，この DMI_suspend() 関数が呼び出されたあとに呼び出される DMI_wake() 関数にしか反応しないことに注意する．つまり，条件変数とは異なり，タイミングの問題で DMI_suspend() 関数が DMI_wake() 関数よりも早く呼ばれてしまった場合には，DMI_suspend() 関数は次の DMI_wake() 関数が呼ばれるまで返らない．使用頻度は少ない.
- int32_t DMI_self(DMI_thread_t *dmi_thread_ptr)
  この DMI_self() 関数を呼び出した DMI スレッドのハンドルを dmi_thread_ptr に格納する.

■ 2-14 その他の API

その他には以下の API がある．これらは使用頻度が低かったり，試験的な実装しか行っていなかったりするため，簡単な説明を付けるにとどめる.

- int32_t DMI_welcome(int32_t dmi_id)
  ID が dmi_id の DMI プロセスを参加させる.
- int32_t DMI_goodbye(int32_t dmi_id)
  ID が dmi_id の DMI プロセスを脱退させる.
- int32_t DMI_poll(DMI_node_t *dmi_node_ptr)
  系内に生じている参加/脱退要求をポーリングする．参加/脱退要求が生じるまで待機したあと，参加/脱退要求を出している DMI プロセスの情報が dmi_node_ptr に格納される.
- int32_t DMI_peek(DMI_node_t *dmi_node_ptr, int8_t *flag_ptr)
  系内に生じている参加/脱退要求をポーリングする．参加/脱退要求を出している DMI プロセスの情報が dmi_node_ptr に格納される．DMI_poll() と異なり，この API はすぐに返る．参加/脱退要求が生じていれば flag_ptr に TRUE が，生じていなければ FALSE が格納される.
- int32_t DMI_fork(char *option, int32_t dmi_id, DMI_node_t *dmi_node_ptr)
  ID が dmi_id の DMI プロセスと同一のノードに新しい DMI プロセスを生成する．生成した DMI プロセスの情報が dmi_node_ptr に格納される．DMI プロセスを生成する場合のコマンドラインオプションを option に指定できる.
- int32_t DMI_kill(int32_t dmi_id)

ID が dmi_id の DMI プロセスに終了通知を送る．Ctrl+C を送ったのと同様の効果を持つ．

- int32_t DMI_scheduler_init(int64_t dmi_scheduler_addr)
  透過的なスレッド移動を実現するためのスレッドスケジューラ dmi_scheduler_addr を初期化する．dmi_scheduler_addr はグローバルアドレスで，DMI_scheduler_t のサイズを持っていなければならない．
- int32_t DMI_scheduler_destroy(int64_t dmi_scheduler_addr)
  dmi_scheduler_addr で表されるスレッドスケジューラを破棄する．
- int32_t DMI_scheduler_create(int64_t dmi_scheduler_addr, int32_t *sthread_id_ptr, int64_t dmi_addr)
  DMI スレッドを生成して，dmi_scheduler_addr で表されるスレッドスケジューラにその DMI スレッドを登録する．DMI スレッド生成時には，任意のグローバルアドレス dmi_addr を渡すことができる．生成した DMI スレッドのスレッド ID が sthread_id_ptr に格納される．
- int32_t DMI_scheduler_detach(int64_t dmi_scheduler_addr, int32_t sthread_id)
  スレッド ID が sthread_id_ptr の DMI スレッドを，dmi_scheduler_addr で表されるスレッドスケジューラからディタッチする．
- int32_t DMI_scheduler_join(int64_t dmi_scheduler_addr, int32_t sthread_id, int64_t *dmi_addr_ptr)
  スレッド ID が sthread_id_ptr の DMI スレッドを，dmi_scheduler_addr で表されるスレッドスケジューラから回収する．
- int32_t DMI_yield(void)
  スレッドスケジューラがこの DMI スレッドを移動するチャンスを与える．
- void* DMI_thread_mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
  DMI スレッド固有のメモリ領域を mmap する．
- void* DMI_thread_mremap(void *old_address, size_t old_size, size_t new_size, int flags)
  DMI スレッド固有のメモリ領域を mremap する．
- int32_t DMI_thread_mprotect(void *addr, size_t len, int prot)
  DMI スレッド固有のメモリ領域を mprotect する．
- int32_t DMI_thread_munmap(void *start, size_t length)
  DMI スレッド固有のメモリ領域を munmap する．
- void* DMI_thread_malloc(int64_t size)
  DMI スレッド固有のメモリ領域を malloc する．
- void* DMI_thread_realloc(void *ptr, int64_t size)
  DMI スレッド固有のメモリ領域を realloc する．
- void DMI_thread_free(void *ptr)
  DMI スレッド固有のメモリ領域を free する．

- `int32_t DMI_idpool_init(int64_t dmi_idpool_addr, int32_t id_max)`
  グローバルアドレス空間に idpool を作成する．`dmi_idpool_addr` はグローバルアドレスで，`DMI_idpool_t` のサイズを持っていなければならない．`id_max` はこの idpool で使える id の最大値である．idpool とは，0 以上 `id_max` 未満の id を管理していて，get したときに，その時点で使われていない id のうちできるだけ小さい値（最小値とはかぎらない）を返してくれるようなデータ構造である．
- `int32_t DMI_idpool_destroy(int64_t dmi_idpool_addr)`
  `dmi_idpool_addr` で表される idpool を破棄する．
- `int32_t DMI_idpool_get(int64_t dmi_idpool_addr, int32_t *id_ptr)`
  `dmi_idpool_addr` で表される idpool から id を取得する．
- `int32_t DMI_idpool_put(int64_t dmi_idpool_addr, int32_t id)`
  `dmi_idpool_addr` で表される idpool に id を返却する．

参考文献

[1] 原健太朗．再構成可能な高性能並列計算のための PGAS プログラミング処理系．修士論文．2011/2