

DMI ドキュメント（執筆途中）

A Global Address Space Framework
for Elastic and High-performance Parallel Computations

原健太郎

<http://haraken.info/>

2010.2.22

Contents

Contents	2
Chap.1 DMI について	6
1 DMI とは何か	6
1-1 超要約	6
1-2 長期的な目標	6
1-3 さしあたっての目標	7
2 本ドキュメントについて	8
2-1 本ドキュメントの目的	8
2-2 対象とする読者	8
2-3 本ドキュメントの構成	8
3 ライセンス	9
4 お知らせ	9
5 インストール	10
5-1 動作環境	10
5-2 インストール	10
5-3 コンパイル	11
5-4 実行	11
Chap.2 DMI プログラミングの基礎	15
1 DMI プログラミングの具体例	15
1-1 行列行列積	15
1-2 プログラム例	16
1-3 プログラム解説	19
2 共通のルール	19
2-1 DMI プログラミングのテンプレート	19
2-2 変数名の命名規則	20
2-3 関数の返り値	20
2-4 非同期 API	20
2-5 グローバル構造体とローカル構造体	21
Chap.3 プロセスとスレッド	23
1 基本事項	23
1-1 DMI プロセスと DMI スレッド	23

1-2	DMI プログラミングにおける実行モデル	23
2	プロセスの参加/脱退	24
2-1	dmirun コマンド	25
2-1-1	参加通知の送り方	25
2-1-2	脱退通知の送り方	25
2-1-3	進捗状況の表示	26
2-1-4	コマンドラインオプション	27
2-1-5	並列シェルを使って一括して参加/脱退させる	28
2-2	dmimw コマンド	29
2-2-1	概要	29
2-2-2	参加通知の送り方	29
2-2-3	脱退通知の送り方	30
2-2-4	ノードファイルのフォーマット	30
2-2-5	コマンドラインオプション	31
2-3	参加/脱退通知の検知と許可	32
2-3-1	概要	32
2-3-2	参加/脱退通知を検知する API	32
2-3-3	DMI_node_t 構造体	32
2-3-4	参加/脱退を許可する API	33
2-3-5	具体例	34
2-3-6	その他の API	36
3	スレッドの生成/回収	37
3-1	スレッドの生成/回収/detach	37
3-2	具体例	38
3-3	補足	40
3-3-1	DMI_main(...) を途中で脱退させることは可能か	40
Chap.4	メモリアクセス	41
1	基本事項	41
1-1	ローカルメモリとグローバルメモリ	41
1-2	プログラミングの基本指針	42
2	メモリの確保/解放	43
2-1	ページ	43
2-2	API	43
2-3	補足	44
2-3-1	ページの実体が確保されるタイミング	44
2-3-2	DMI_mmap(...) / DMI_munmap(...) は重い	45
2-3-3	DMI_mmap(...) / DMI_munmap(...) を行うタイミング	45

2-3-4	構造体の特定のメンバへの read/write	45
3	グローバルメモリの read/write	45
3-1	API	45
3-1-1	API	45
3-1-2	高性能なプログラムを記述するうえでのポイント	46
3-2	コンシステンシモデル	47
3-2-1	直観的な説明	47
3-2-2	正確な説明	48
3-2-3	複数のページにまたがる DMI_read(...)/DMI_write(...)	48
3-3	選択的キャッシュ read/write	49
3-3-1	キャッシュ管理の仕組み	49
3-3-2	選択的キャッシュ read/write の詳細	50
3-3-3	mode を選択する際の一般論	51
3-3-4	具体例 (遠隔スワップ)	52
3-3-5	具体例 (行列行列積)	54
3-4	より高度なチューニング	56
3-4-1	必要なページをキャッシュから落とさない	56
3-4-2	データ転送の負荷を分散する	57
3-5	補足	58
3-5-1	非同期 read/write に与えるローカルメモリを再利用して良いタイミング	58
3-5-2	グローバル変数は使用できない	58
3-5-3	スレッドローカルストレージは使用できない	59
4	離散的な read/write	60
4-1	概要	60
4-2	API	60
4-3	具体例	61
4-4	補足	61
4-4-1	離散的な read/write の性能	61
5	read-write-set	62
5-1	ローカル不透明オブジェクトとグローバル不透明オブジェクト	62
Chap.5	同期	63
1	排他制御変数	63
1-1	API	63
1-2	具体例	64
2	条件変数	66
2-1	API	66
2-2	具体例	66

3	バリア操作	69
3-1	API	69
3-2	具体例	70
3-3	補足	71
3-3-1	DMI_local_barrier_allreduce(...) が返るタイミング	71
4	スレッドスケジューリング	72
4-1	API	72
5	組み込みの read-modify-write	72
5-1	read-modify-write	72
5-1-1	read-modify-write	72
5-1-2	compare-and-swap	73
5-1-3	fetch-and-store	74
5-2	API	74
6	ユーザ定義の read-modify-write	75
6-1	概要	75
6-2	API	76
6-2-1	簡単化した説明	76
6-2-2	正確な説明	77
6-3	具体例	78
6-3-1	カウンタ変数をアトミックにインクリメントする	78
6-3-2	Allreduce	78
6-4	補足	78
6-4-1	DMI における同期の各種手段の性能比較	78
Chap.6 サンプルプログラム		79
1	典型的な DMI プログラミング	79
1-1	DMI プロセスが動的に参加/脱退するプログラム	79
1-2	SPMD 型のプログラム	79
1-3	pthread プログラムから DMI プログラムへの変換	79
2	サンプルプログラムの実行方法	79
Chap.7 API 一覧		80



Chap.1 DMI について

1 DMI とは何か

■ 1-1 超要約

DMI (Distributed Memory Interface) は、クラウドコンピューティングの世界において高性能並列計算を簡単に記述できるようにすることを目指した並列分散プログラミング処理系です。「クラウドコンピューティング」というバズワードを使わずに表現するならば、DMI では、Fig.??のように、並列計算を実行中に利用できる計算資源数が動的に増減するような環境において、計算規模を動的に拡張/縮小しながら動作するような高性能並列計算を簡単に記述することができます。

■ 1-2 長期的な目標

では、詳しく説明していきます。クラウドコンピューティングでは、クラウドプロバイダと呼ばれる組織が大規模なデータセンタを構築し、インフラストラクチャ、プラットフォーム、ソフトウェアなどを整備して、それらをサービスとして利用者に提供します。そして、利用者は、それらのサービスを必要なときに必要な量だけ利用することができ、実際に利用した量だけの課金がなされます。したがって、従来ならば、企業や大学が何らかの大規模な計算を行うためには、必要な計算規模を予測したうえで自前でデータセンタを構築して管理する必要があったのに対して、クラウドコンピューティングでは、面倒なデータセンタの構築や管理の手間をすべてクラウドプロバイダに任せられるうえ、必要なときに必要なだけの計算規模を利用できるため、多くの場合にはコストパフォーマンスが良くなります。つまり、クラウドコンピューティングは、計算環境を「所有」する従来の形態を、計算環境を「利用」する形態へとパラダイムシフトさせたわけです。

クラウドコンピューティングにはさまざまな目的や形態が存在しますが、DMI は特に、クラウドコンピューティングの世界で高性能並列計算を効率良く実行できるようなプラットフォームをどう実現するか、という問題に着眼しています。今述べたように、クラウドコンピューティングの世界では、利用者側の視点で見れば、何か 1 個の大規模なデータセンタとその上で動く並列計算用プラットフォームがあってそれ

を必要なときに必要なだけ利用できるわけですが、これをクラウドプロバイダ側の視点で見ると、データセンタを利用している利用者が多数いて、多数の利用者が好き勝手に要求してくる並列計算を、有限の計算資源をうまくやり繰りしながら効率良く処理していくことが必要になります。さて、ここで有限の計算資源をうまくやり繰りしながら効率良く並列計算を実行するにはどうすれば良いかを考えてみます。たとえば、ある時点で利用者 A が並列計算を要求してきたときに、データセンタを利用している利用者は他には誰もいなかったとします。このときは、Fig.??に示すように、データセンタに存在する計算資源をフルに使って利用者 A の並列計算を実行するのが効率的だと考えられます。しばらくして、利用者 B も並列計算を要求してきたとします。このとき、すでに利用者 A の並列計算がすべての計算資源を使い切っているため、このままでは利用者 B の並列計算を実行することができません。このような場合には、Fig.??に示すように、すでに長時間実行している利用者 A の並列計算の規模を縮小して、空いた分の計算資源に利用者 B の並列計算を割り当てるのが効率的だと考えられます。当然、複数の利用者による並列計算の要求をどのようにスケジューリングするかは運用ポリシーにも依存する難しい問題ですが、少なくとも、データセンタの中の計算資源が有限である限り、並列計算を実行中にその並列計算の計算規模を動的に拡張/縮小させる必要が出てきます。

ところが、少し想像すればわかるように、実行途中で計算規模が拡張/縮小するような並列計算のプログラムを記述するのは、到底容易なことではありません。実際に、MPI^{*1}、UPC^{*2}、Global Arrays^{*3}、Titanium^{*4} など、高性能並列計算のための並列分散プログラミング処理系は多数開発されていますが、並列計算の計算資源を動的に拡張/縮小させることができる処理系はほとんど存在しません。Phoenix^{*5} というメッセージパッシングベースの処理系では、並列計算の計算資源を動的に拡張/縮小させることができますが、メッセージパッシングベースであるということもあり、プログラミングは複雑なものになっています。

以上の動機に基づき、DMI では、グローバルアドレス空間を提供し、簡単な read/write ベースのプログラミングによって、計算規模が動的に拡張/縮小するような高性能並列計算を簡単に記述できることを目標としています。

■ 1-3 さしあたっての目標

このように、DMI では、クラウドコンピューティングの世界で高性能並列計算を効率良く実行できるようなプラットフォームを実現することを長期的な視野として目標にしています。しかし、実際にクラウドコンピューティングとしてのプラットフォームを作り上げるには、ユーザインタフェース、セキュリティ、SLA などあまりに多様な要素を考慮する必要があり、実用的に運用可能なシステムまで作り込むのは非常に困難です。そこで、DMI では、さしあたって、1 つの均質なクラスタ環境において、ノードを動的に参加/脱退させることで計算規模が動的に拡張/縮小するような高性能並列計算を簡単に記述できることを目標にします。そのために、DMI では、

*1

*2

*3

*4

*5

- ノードの動的な参加/脱退を簡単に記述するための API
- スレッド生成/回収，グローバルアドレス空間へのメモリ確保/解放，グローバルアドレス空間に対する read/write，排他制御変数や条件変数による同期など，並列計算を記述するうえで必要となる一連の API
- データの所在を明示的かつ細粒度にコントロールすることで並列計算の性能を徹底的に最適化するための手段

などを提供しています．また，DMI は計算規模の動的な拡張/縮小を大きなテーマとしていますが，当然，実行開始から実行終了まで計算資源数が一定であるような高性能並列計算のためにも利用できます．DMI では，計算規模の拡張/縮小にかかわらず，グローバルアドレス空間に対する read/write に基づいた容易なプログラミングと明示的で強力な最適化手段のもとで，さまざまな高性能数値計算を見通し良く開発することができます．

2 本ドキュメントについて

■ 2-1 本ドキュメントの目的

本ドキュメントの目的は，はじめて DMI を使う人が，効率的な DMI プログラミングを行えるようになるために必要十分な情報をチュートリアル形式で提供することにあります．よって，単なる「仕様書」ではなく，DMI のインストール方法，実行方法，DMI プログラミングのルール，DMI の各 API の詳細なセマンティクス，DMI プログラムのチューニング方法などを，できるだけ論理的に飛躍することなく順を追って解説することを目指しています．特に，分散プログラミングにおいては，いかにしてスケラビリティを高めるかが問題になるため，重要な部分については DMI の内部的な実装まで話を踏み込んで，DMI プログラムを性能最適化するためのヒントを詳しく解説しています．

しかし，あくまでもこのドキュメントは，プログラマが DMI プログラミングを行うためのドキュメントであるため，DMI の処理系の概念，意義，モデル，詳細なアルゴリズムなど DMI の詳細な実装については説明していません．それら学術的な内容に関しては，DMI の配布サイトにある各論文をご参照ください．

■ 2-2 対象とする読者

おおむね，以下のようなスキルの読者を対象に解説しています：

- C 言語のプログラミングに十分慣れている．
- MPI や UPC などでも並列分散プログラムを少しは書いた経験がある．
- pthread プログラミングや同期に関する知識がある．
- Linux での日常的なシェル操作ができる．

■ 2-3 本ドキュメントの構成

本ドキュメントは以下のように構成されています：

第 1 章 DMI について DMI の紹介，インストール方法，実行方法，ライセンスなどメタ的な説明を行います．

第 2 章 DMI プログラミングの基礎 実際の DMI プログラムを提示し，おおよそ DMI プログラムがどのように記述できるのかを解説するとともに，DMI プログラミングにおける最も基本的なルールを説明します．

第 3 章 プロセスとスレッド DMI プログラミングにおいて，動的に DMI プロセスを参加/脱退させたり DMI スレッドを生成/回収したりすることで，計算規模を拡張/縮小するための方法を説明します．

第 4 章 メモリアクセス グローバルアドレス空間に対するメモリ確保/解放やさまざまな read/write の方法を説明するとともに，DMI の内部的な実装にも踏み込み，DMI プログラムの性能を最適化する方法について詳しく説明します．

第 5 章 同期 DMI が提供するさまざまな同期の方法について説明します．

第 6 章 サンプルプログラム 典型的な DMI プログラミングのテンプレートを紹介するとともに，`dmi-x.x.x.x/sample/` ディレクトリ以下に入っているサンプルプログラムの概要と実行方法を説明します．

第 7 章 API 一覧 以上の各章で断片的に説明してきた API を一覧にしてまとめています．

3 ライセンス

本ソフトウェアのライセンスは以下のとおりです：

- 本ソフトウェアの著作権は原健太郎にあります．
- 現時点での DMI の最新バージョンは，DMI 1.3.0 です．
- 本ソフトウェアは GPL ライセンスに従います．

4 お知らせ

DMI は，クラウドな並列計算を容易に記述できるようにすることを目的として研究開発されています．現段階では，実用性よりも研究レベルでの新規性を重視しているため，現時点で最も「概念的」に美しいソフトウェアにすることを最重視して開発しています．よって，後方互換性は意識しておらず予告なく API の変更を行うことがありますのでご了承ください．

DMI に関する更新履歴や最新情報は，DMI の配布サイトに掲載しますのでご参照ください．

バグ報告，DMI に関する質問，意見，ドキュメントの誤植報告などは大歓迎です（特に DMI の設計に関する意見は大歓迎です）．DMI の配布サイトにある掲示板に書き込むか，もしくは私のメールアドレス：

haraken ♡ logos.ic.i.u-tokyo.ac.jp（♡ の部分を@に置き換えてください）

までメールしていただければ幸いです．

5 インストール

■ 5-1 動作環境

64 ビットの Linux 環境で動作します。64 ビットであることを陽に利用しているため、32 ビットの Linux 環境では安全に動作しません。ソフトウェアとしては、gcc コンパイラおよび Perl を必要とします。

DMI の開発では、主に以下の環境を利用して動作確認を行っています：

- Intel Xeon E5410 (4 コア) 2.33GHz を 2 個搭載した 8 コアマシン 16 台を 1GbitEthernet で接続したクラスター。OS はカーネル 2.6.18-6-amd64 の Debian Linux。gcc のバージョンは 4.1.2。
- Intel Xeon 5140 (4 コア) 2.33GHz を 1 個搭載した 4 コアマシン 22 台を 1GbitEthernet で接続したクラスター。OS はカーネル 2.6.18-6-amd64 の Debian Linux。gcc のバージョンは 4.1.2。
- AMD Opteron 8356 (4 コア) 2.30GHz を 4 個搭載した 16 コアマシン 8 台を、Myrinet および 1GbitEthernet で接続したクラスター。OS はカーネル 2.6.18-53.1.19.el5 の RedHatEnterprise Linux。gcc のバージョンは 4.1.2。

■ 5-2 インストール

インストールの手順を説明します。ここではクラスター環境を想定して、ホームディレクトリは NFS などのファイル共有システムによってファイル共有されているとし、DMI をホームディレクトリ直下の /dmi/ ディレクトリにインストールする場合を例にして説明します。なお、ファイル共有されていればインストール作業や DMI プログラムの実行が簡単になりますが、DMI の実行にとってファイル共有されていることは必須ではありません。ファイル共有されていなくても、DMI がインストールされていてかつ DMI プログラムの実行バイナリを保持していれば、その DMI プログラムに基づく並列計算に参加/脱退させることができます。

dmi-x.x.x.x.tar.gz をダウンロードしたディレクトリに移動し、以下のコマンドを入力することで、/dmi/ディレクトリに DMI をインストールすることができます。なお、dmi-x.x.x.x の部分は DMI のバージョンに応じて読み替えてください：

```
$ tar xvf dmi-x.x.x.x.tar.gz
$ cd dmi-x.x.x.x/
$ ./configure --prefix=~/.dmi/
$ make
$ make install
```

次に、/dmi/bin ディレクトリにパスを通すために以下のコマンドを打ちます：

```
$ export PATH=$PATH:~/.dmi/bin
```

これで dmicc、dmirun、dmimw の各コマンドが利用できるようになります。さらに、次回以降の口

グインしたときに自動的にパスが通るようにするために、`.bashrc` を編集して次のコマンドを書いた行を追加してください：

```
export PATH=$PATH:~/dmi/bin
```

今はホームディレクトリ以下がファイル共有されていることを仮定しているため、以上の作業で、クラスタ内のすべてのノードで DMI がインストールされパスが設定されたことになります。ファイル共有されていない場合、すべてのノードで上記の手順によって DMI のインストールとパスの設定を行ってください。

■ 5-3 コンパイル

`dmi-x.x.x.x/sample/` ディレクトリの中には、サンプルの DMI プログラムがいくつか入っています。ここでは `sample` ディレクトリの中の `mandel_dmi.c` をコンパイルして実行してみます。これはマンデルブロ集合を並列描画する DMI プログラムです。ここではコンパイルから実行までの流れの概略のみ説明しますので、この DMI プログラムの詳細については??節を、実行方法の詳細については??を参照してください。

まず、`dmi-x.x.x.x/sample/` ディレクトリに移動します：

```
$ cd dmi-x.x.x.x/sample/
```

コンパイルは以下のコマンドで行います：

```
$ dmicc -O3 mandel_dmi.c -o mandel_dmi
```

これで実行バイナリ `mandel_dmi` が得られます。`dmicc` コマンドには、`gcc` のコンパイルオプションを任意に指定できます。ここでは最適化オプションの `-O3` を指定しています。

なお、`sample` ディレクトリで、

```
$ make
```

と打つと、すべてのサンプルプログラムをコンパイルすることができます。

今はホームディレクトリ以下がファイル共有されていることを仮定しているため、以上の作業で、クラスタ内のすべてのノードで実行バイナリ `mandel_dmi` が生成されたことになります。ファイル共有されていない場合、すべてのノードでコンパイルを行って実行バイナリ `mandel_dmi` を作ってください。

■ 5-4 実行

`kototoi000`、`kototoi001` というホスト名を持つ 2 つのノードで `mandel_dmi` を実行して並列計算を行うとします。わかりやすいように、ターミナルを 2 個起動してください*⁶。

*⁶ 以下の作業は、それなりにテンポ良く行ってください。そうでないと、いろいろコマンドを入力してノードの参加/脱退させる前に、並列計算の方が終了してしまいます。

まず、ターミナル 1 から kototoi000 にログインして、コマンドラインから、

```
$ dmirun ./mandel_dmi 480 480 480 1000000 0
=== initialized ===
=== joined ===
press any key to continue ... # ここでも Enter キーを入力する
```

と入力すると、480 個のタスクが生成されて実行が始まります。すると DMI プログラムが実行され始め、ターミナル 1 では以下のような出力が見えるはずです：

```
$ ./dmirun ./mandel_dmi 480 480 480 1000000 0
=== initialized ===
=== joined ===
press any key to continue ...

welcome 0!
=== opened ===
started
started
started
started
started
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
44 45 46 started
started
47 48 50 49 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 78 77 7
```

出力されている数字が、処理されたタスクの番号です。479 番が一番最後のタスクになります。

次に、ターミナル 2 から kototoi001 にログインして、コマンドラインから、

```
$ dmirun -i kototoi000 ./mandel_dmi
```

と入力すると、kototoi001 を先ほど kototoi000 で実行した DMI プログラムに参加させ、並列計算の計算規模が拡張させることができます。このときターミナル 2 では、

```
$ dmirun -i kototoi000 ./mandel_dmi
=== initialized ===
=== joined ===
started
started
started
started
started
started
=== opened ===
started
started
127 128 129 130 131 132 133 134 143 144 145 146 147 148 150 151 159 160 161 162 163 164 166 167 1
```

のような kototoi001 からの出力が見えるはずです。

さて、このときターミナル 1 とターミナル 2 では、それぞれ kototoi000 と kototoi001 からの出力が続いており、並列計算が実行され続けているのがわかりますが、このとき、ターミナル 2 で Ctrl+C を 1 回入力すると、kototoi001 を並列計算から脱退させ、並列計算の計算規模を縮小させることができます。実際にターミナル 2 で Ctrl+C を入力すると、

```
$ dmirun -i kototoi000 ./mandel_dmi
=== initialized ===
=== joined ===
started
started
started
started
started
started
started
=== opened ===
started
started
127 128 129 130 131 132 133 134 143 144 145 146 147 148 150 151 159 160 161 162 163 164 166 167 1
ここで Ctrl+C を入力
=== closed ===
finished
finished
finished
finished
finished
finished
finished
finished
finished
finished
=== left ===
=== finalized ===
```

のような出力が得られ、kototoi001 のコマンドラインが返ってきます。一方、ターミナル 1 では依然として、

```
なお、
$ ./dmirun ./mandel_dmi 480 480 480 1000000 0
=== initialized ===
=== joined ===
press any key to continue ...

welcome 0!
=== opened ===
started
started
started
started
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 started
started
38 started
40 39 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 71 7
135 136 137 138 139 140 141 142 149 152 153 154 155 156 157 158 165 168 169 170 171 172 173 174 1
```

```
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 2
```

のような出力が続いており，kototoi001 が脱退しても依然として計算が続いているのがわかります．しばらくすると kototoi001 でも計算が終了してコマンドラインが返ってきます．

以上が，DMI において動的にノードを参加/脱退させる際のおおまかな実行の流れになります．



Chap.2 DMI プログラミングの基礎

本章では，行列行列積のサンプルプログラムを題材として，DMI プログラミングがおおよそどのようなものなのかを概観します．詳細な説明はすべて次章以降に回し，本章では，DMI プログラミングの雰囲気と DMI プログラミングにおける共通のルールを説明します．

1 DMI プログラミングの具体例

■ 1-1 行列行列積

本節では， $pnum$ 個の DMI スレッドを使って行列行列積を並列に計算する DMI プログラムを題材にして，DMI プログラミングの概観を説明します．DMI の最大の目的は，計算規模が動的に拡張/縮小するような並列プログラムを容易に記述可能にすることにあります，ここでは話を簡単化するために，DMI プログラムの実行開始から実行終了まで，計算規模を変化させることなく一定数の DMI スレッドで並列に行列行列積を計算する DMI プログラムを題材にします．

解くべき問題は， $n \times n$ のサイズの行列 A, B が与えられたとき，行列行列積 $AB = C$ を求めることです．アルゴリズムとしては，簡単な横ブロック分割のアルゴリズムを用います：

- (1) まず，DMI スレッド 0 で行列 A, B を初期化します．
- (2) Fig.??に示すように，行列 A を横方向に $pnum$ 等分します．これらの部分行列を $A_0, A_1, \dots, A_{pnum-1}$ とします．DMI スレッド 0 は，部分行列 $A_0, A_1, \dots, A_{pnum-1}$ と行列 B をグローバルアドレス空間に書き込みます (Fig.??(A))．
- (3) 各 DMI スレッド i は，グローバルアドレス空間から A_i と B を読んで，部分行列行列積 $A_i B = C_i$ を計算したあと，グローバルアドレス空間に C_i を書き込みます (Fig.??(B))．
- (4) DMI スレッド 0 は，グローバルアドレス空間から行列 C を読み出します (Fig.??(C))．

■ 1-2 プログラム例

以上で説明した行列行列積は、DMI で以下のように記述できます。各 API のセマンティクスや各用語の意味は次章以降で詳しく解説するので、ここでは雰囲気だけをつかんでください：

```
#include "dmi_api.h"

typedef struct targ_t
{
    int32_t rank; /* DMI スレッドのランク */
    int32_t pnum; /* DMI スレッドの個数 */
    int32_t n; /* 行列のサイズ */
    int64_t a_addr; /* 行列 A のグローバルメモリ */
    int64_t b_addr; /* 行列 B のグローバルメモリ */
    int64_t c_addr; /* 行列 C のグローバルメモリ */
    int64_t barrier_addr; /* バリア操作のためのグローバル不透明オブジェクト */
}targ_t;

double sumof_matrix(double *matrix, int32_t n);

void DMI_main(int argc, char **argv)
{
    DMI_node_t node;
    DMI_node_t *nodes;
    DMI_thread_t *threads;
    targ_t targ;
    int32_t i, j, n, node_num, thread_num, rank, pnum;
    int64_t targ_addr, a_addr, b_addr, c_addr, barrier_addr;

    if(argc != 4)
    {
        fprintf(stderr, "usage : %s node_num thread_num n\n", argv[0]);
        exit(1);
    }
    node_num = atoi(argv[1]); /* 実行する DMI プロセスの個数 */
    thread_num = atoi(argv[2]); /* 各 DMI プロセスに生成する DMI スレッドの個数 */
    n = atoi(argv[3]); /* 行列のサイズ */
    pnum = node_num * thread_num; /* 生成する DMI スレッドの個数 */

    nodes = (DMI_node_t*)my_malloc(node_num * sizeof(DMI_node_t));
    threads = (DMI_thread_t*)my_malloc(pnum * sizeof(DMI_thread_t));
    for(i = 0; i < node_num; i++) /* すべての DMI プロセスを参加させる */
    {
        DMI_poll(&node); /* DMI プロセスの参加/脱退通知を検知 */
        if(node.state == DMI_OPEN) /* 参加通知ならば */
        {
            DMI_welcome(node.dmi_id); /* その DMI プロセスの参加を許可 */
            nodes[i] = node;
        }
    }

    DMI_mmap(&targ_addr, sizeof(targ_t), pnum, NULL); /* DMI スレッドに渡す引数を格納するグローバルメモリを確保 */
    DMI_mmap(&barrier_addr, sizeof(DMI_barrier_t), 1, NULL); /* バリア操作のためのグローバル不透明オブジェクトのグローバルメモリを確保 */
```



```

    DMI_mmap(&a_addr, n / pnum * n * sizeof(double), pnum, NULL); /* 行列 A のグローバルメモリを確保 */
    DMI_mmap(&b_addr, n * n * sizeof(double), 1, NULL); /* 行列 B のグローバルメモリを確保 */
    DMI_mmap(&c_addr, n / pnum * n * sizeof(double), pnum, NULL); /* 行列 C のグローバルメモリを確保 */
    DMI_barrier_init(barrier_addr); /* バリア操作のためのグローバル不透明オブジェクトの初期化 */
    for(rank = 0; rank < pnum; rank++) /* 各 DMI スレッドに渡す引数をグローバルメモリに仕込む */
    {
        targ.rank = rank;
        targ.pnum = pnum;
        targ.n = n;
        targ.a_addr = a_addr;
        targ.b_addr = b_addr;
        targ.c_addr = c_addr;
        targ.barrier_addr = barrier_addr;
        DMI_write(targ_addr + rank * sizeof(targ_t), sizeof(targ_t), &targ, DMI_EXCLUSIVE_WRITE, NULL);
    }
    rank = 0;
    for(i = 0; i < node_num; i++) /* 参加中のすべての DMI プロセスに関して */
    {
        node = nodes[i];
        for(j = 0; j < thread_num; j++) /* 各 DMI プロセスあたり thread_num 個の DMI スレッドを生成 */
        {
            DMI_create(&threads[rank], node.dmi_id, targ_addr + rank * sizeof(targ_t), NULL);
            rank++;
        }
    }
    for(rank = 0; rank < pnum; rank++) /* すべての DMI スレッドを回収 */
    {
        DMI_join(threads[rank], NULL, NULL);
    }
    DMI_barrier_destroy(barrier_addr); /* バリア操作のためのグローバル不透明オブジェクトを破棄 */
    DMI_munmap(c_addr, NULL); /* 行列 c のグローバルメモリを解放 */
    DMI_munmap(b_addr, NULL); /* 行列 B のグローバルメモリを解放 */
    DMI_munmap(a_addr, NULL); /* 行列 A のグローバルメモリを解放 */
    DMI_munmap(barrier_addr, NULL); /* バリア操作のためのグローバル不透明オブジェクトのグローバルメモリを解放 */
    DMI_munmap(targ_addr, NULL); /* 各 DMI スレッドに渡す引数を格納するためのグローバルメモリを解放 */
    for(i = 0; i < node_num; i++) /* すべての DMI プロセスを脱退させる */
    {
        DMI_poll(&node); /* DMI プロセスの参加/脱退通知を検知 */
        if(node.state == DMI_CLOSE) /* 脱退通知ならば */
        {
            DMI_goodbye(node.dmi_id); /* その DMI プロセスの脱退を許可 */
        }
    }
    my_free(threads);
    my_free(nodes);
    return;
}

/* 各 DMI スレッドが実行する関数 */
int64_t DMI_thread(int64_t targ_addr)
{
    int32_t my_rank, pnum, i, j, k, n;

```

```

int64_t a_addr, b_addr, c_addr, barrier_addr;
targ_t targ;
double *original_a, *original_b, *original_c, *local_a, *local_b, *local_c;
double dummy;
DMI_local_barrier_t local_barrier;

DMI_read(targ_addr, sizeof(targ_t), &targ, DMI_GET_READ, NULL); /* 引数を読む */
my_rank = targ.rank; /* この DMI スレッドのランク */
pnum = targ.pnum; /* DMI スレッドの個数 */
n = targ.n; /* 行列のサイズ */
a_addr = targ.a_addr; /* 行列 A のグローバルメモリ */
b_addr = targ.b_addr; /* 行列 B のグローバルメモリ */
c_addr = targ.c_addr; /* 行列 C のグローバルメモリ */
barrier_addr = targ.barrier_addr; /* バリア操作のためのグローバル不透明オブジェクト */
DMI_local_barrier_init(&local_barrier, barrier_addr); /* バリア操作のためのローカル不透明オブジェクトを初期化 */

local_a = (double*)my_malloc(n / pnum * n * sizeof(double));
local_b = (double*)my_malloc(n * n * sizeof(double));
local_c = (double*)my_malloc(n / pnum * n * sizeof(double));
for(i = 0; i < n / pnum; i++)
{
    for(j = 0; j < n; j++)
    {
        local_c[i * n + j] = 0;
    }
}
if(my_rank == 0) /* 行列 A, B を初期化 */
{
    original_a = (double*)my_malloc(n * n * sizeof(double));
    original_b = (double*)my_malloc(n * n * sizeof(double));
    original_c = (double*)my_malloc(n * n * sizeof(double));
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
        {
            original_a[i * n + j] = 1;
            original_b[i * n + j] = 1;
            original_c[i * n + j] = 0;
        }
    }
    DMI_write(a_addr, n * n * sizeof(double), original_a, DMI_EXCLUSIVE_WRITE, NULL); /* 行列 A のグローバルメモリを初期化 */
    DMI_write(b_addr, n * n * sizeof(double), original_b, DMI_EXCLUSIVE_WRITE, NULL); /* 行列 B のグローバルメモリを初期化 */
}
DMI_local_barrier_allreduce(&local_barrier, 0, &dummy, pnum); /* バリア操作 */

DMI_read(a_addr + my_rank * n / pnum * n * sizeof(double), n / pnum * n * sizeof(double), local_a, /* DMI スレッドがグローバルメモリから部分行列 Ai を読む */
DMI_read(b_addr, n * n * sizeof(double), local_b, DMI_INVALIDATE_READ, NULL); /* 各 DMI スレッドがグローバルメモリから行列 B 全体を読む */
for(i = 0; i < n / pnum; i++) /* 部分行列行列積 Ci=AiB を計算 */
{
    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];
            }
        }
    }
    DMI_write(c_addr + my_rank * n / pnum * n * sizeof(double), n / pnum * n * sizeof(double), local_c);
    /* DMI スレッドが部分行列 Ci をグローバルメモリに書く */
    DMI_local_barrier_allreduce(&local_barrier, 0, &dummy, pnum); /* バリア操作 */
    if(my_rank == 0) /* 結果の行列 C を得る */
    {
        DMI_read(c_addr, n * n * sizeof(double), original_c, DMI_INVALIDATE_READ, NULL); /* 行
列 C 全体をグローバルメモリから読む */
        my_free(original_c);
        my_free(original_b);
        my_free(original_a);
    }
    my_free(local_a);
    my_free(local_b);
    my_free(local_c);
    return DMI_NULL;
}

```

■ 1-3 プログラム解説

大ざっぱにプログラムを解説します。*** 説明 ***

2 共通のルール

■ 2-1 DMI プログラミングのテンプレート

上記のプログラム例からわかるように、さまざまな API が出てきて複雑ですが、概念的には DMI プログラミングは pthread プログラミングと対応しています。pthread プログラミングと同様に、DMI プログラムの最も基本的なテンプレートは以下のようになります：

```

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

/* DMI プログラムの起動時に 1 回だけ実行される関数 */
void DMI_main(int argc, char **argv)
{
    ...;
}

/* 各 DMI スレッドが実行する関数 */
int64_t DMI_thread(int64_t addr)
{
    ...;
}

```

要点をまとめます：

- ヘッダファイル `dmi_api.h` を include します。
- `void DMI_main(int argc, char **argv)` というプロトタイプの関数を必ず記述します。この `DMI_main(...)` は C 言語の `main(...)` に相当するもので、DMI プログラムを起動したときに最初に実行される関数で、コマンドライン引数の個数を `argc` に、引数のデータを `argv` に取ります。
- `int64_t DMI_thread(int64_t addr)` というプロトタイプの関数を記述します。生成される DMI スレッドはこの `DMI_thread(...)` を実行します。引数の `addr` は DMI スレッドを生成するときに与える引数です。

■ 2-2 変数名の命名規則

DMI では多数の関数と定数と型を提供しています。これらの命名規則は以下のルールに従います：

- DMI が提供するすべての関数と定数と型は、プレフィックス `DMI_` から始まります。
- 関数は `DMI_xxx(...)` の形式の名前をしていて `xxx` は小文字です。
- 定数は `DMI_XXX` の形式の名前をしていて `XXX` は大文字です。
- 型は `DMI_xxx_t` の形式の名前をしていて `xxx` は小文字です。

以降、DMI の関数のことを DMI の API と呼びます。

■ 2-3 関数の戻り値

DMI の API は、すべて `int32_t DMI_xxx(...)` というプロトタイプを持っています。つまり、DMI の API の戻り値は `int32_t` であり、その API が成功したときには `DMI_TRUE` が返り、失敗したときには `DMI_FALSE` が返ります。たとえば、割り当てられていないグローバルメモリアドレスを read/write しようとしたときや、存在もしない DMI スレッドを回収しようとしたときなど、セマンティクスの的に違反となるような API 呼び出しを行ったときに、`DMI_FALSE` が返ります。上記のプログラムでも、本来なら各 API の戻り値が `DMI_TRUE` であることを検査すべきですが、簡略化のため省略しています。

■ 2-4 非同期 API

DMI の API は基本的に同期 API ですが、いくつかの API は非同期 API にすることができます。非同期 API にできる API は、`int32_t DMI_xxx(..., DMI_local_status_t *status)` というプロトタイプを持っていて、最後の引数として `DMI_local_status_t *status` をとる API です。たとえば、グローバルアドレス空間から read を行う API である、

- `int32_t DMI_read(int64_t addr, int64_t size, void *buf, int8_t mode,`

```
DMI_local_status_t *status);
```

は非同期 API にすることができます。

同期 API として使う場合には、最後の引数 `status` として `NULL` を指定します。たとえば、

```
DMI_read(addr, size, buf, mode, NULL);
```

というように呼び出せば、この `DMI_read(...)` は同期 API になります。つまり、グローバルアドレス空間から該当するデータを `read` する操作が完了するまで、この API は返ってきません。

非同期 API として使う場合には、最後の引数 `status` として、`DMI_local_status_t` 型の変数へのポインタを渡します。たとえば、

```
DMI_local_status_t status;  
DMI_read(addr, size, buf, mode, &status);
```

というように呼び出せば、この `DMI_read(...)` は非同期 API になります。つまり、グローバルアドレス空間から該当するデータを `read` する操作が完了するのを待つことなく、この API はすぐに返ってきます。非同期 API の完了を検査したり待機したりするためには以下の API を使います：

- `void DMI_wait(DMI_local_status_t *status, int32_t *ret_ptr);`
`status` が関連付けられている非同期 API が完了するまで待機します。非同期 API の戻り値が `ret_ptr` に格納されます。
- `int32_t DMI_check(DMI_local_status_t *status, int32_t *ret_ptr);`
`status` が関連付けられている非同期 API が完了しているかどうかを検査します。完了していれば `ret_ptr` に非同期 API の戻り値が格納され、`DMI_check(...)` は `DMI_TRUE` を返します。完了していなければ `DMI_check(...)` は `DMI_FALSE` を返し、このときの `ret_ptr` の値は未定義です。`DMI_wait(...)` とは異なり、`DMI_check(...)` はすぐに返ります。

■ 2-5 グローバル構造体とローカル構造体

DMI では、データの共有をグローバルアドレス空間を介して行いますが、このときグローバルアドレス空間を介して他の DMI プロセスに渡したとしても、データとしての「意味が保存される」データと「意味が保存されない」データがあります。たとえば、ある DMI スレッド t_1 が 12345 という整数データをグローバルアドレス空間に書き込み、それを別の DMI プロセスに存在する DMI スレッド t_2 がグローバルアドレス空間から読んだとします。このとき DMI スレッド t_2 にとっても、このデータは 12345 という整数データになっています。つまり、12345 という整数データは、グローバルアドレス空間を介して他の DMI プロセスに渡したとしても意味が保存されます。別の例として、ある DMI スレッド t_1 が適当なファイルをオープンして、`FILE*` 型のファイルポインタ `fp` を扱っていたとします。このとき、DMI スレッド t_1 がグローバルアドレス空間に `fp` から `sizeof(FILE)` バイトを書き込み、それを別の DMI プロセスに存在する DMI スレッド t_2 が読んだとします。このとき DMI スレッド t_2 が、グローバルアドレス空間から読み出したファイルポインタ `fp` を使って、実際にファイル操作ができるかというと、(もちろん

DMI の処理系をどう作るかにもよりますが、おそらく) ファイル操作には失敗すると考えられ、実際に失敗します。これは、ファイルポインタ `fp` というのが各 DMI プロセスに固有のデータであるためです。つまり、ファイルポインタというデータは、グローバルアドレス空間を介して他の DMI プロセスに渡したときに意味が保存されません。このように、データには、グローバルアドレス空間を介して別の DMI スレッドに渡したときに、意味が保存されるデータと意味が保存されないデータがあります。

この区別に対応して、DMI が提供する型にも 2 種類あります。1 つ目は、その型で宣言された変数が、グローバルアドレス空間を介して別の DMI スレッドに渡しても意味を保存するような型です。DMI ではこの種類の型をグローバル構造体と呼び、グローバル構造体で宣言される変数をグローバル不透明オブジェクトと呼びます。グローバル構造体は `DMI_XXX_t` の形式の名前をしていて、`DMI_` の次の部分には `local` 以外の文字列が続きます。2 つ目は、その型で宣言された変数が、グローバルアドレス空間を介して別の DMI スレッドに渡してしまうと意味が保存されないような型です。DMI ではこの種類の型をローカル構造体と呼び、ローカル構造体で宣言される変数をローカル不透明オブジェクトと呼びます。一方、ローカル構造体は `DMI_local_XXX_t` の形式の名前をしています。

ややわかりにくいので具体例を見てみます。たとえば、バリア操作に使う構造体として `DMI_local_barrier_t` という型がありますが、これはローカル構造体です。よって、`DMI_local_barrier_t` 型の変数 `b` があったとき、`b` の先頭アドレスから `sizeof(DMI_local_barrier_t)` バイトをグローバルアドレス空間経由で他の DMI スレッドに渡しても、渡された DMI スレッドは `b` を使うことはできません。`b` はローカル不透明オブジェクトなので意味が保存されないからです。別の例として、同じくバリア操作に使う構造体として `DMI_barrier_t` という型がありますが、これはグローバル構造体です。よって、`DMI_barrier_t` 型の変数 `b` があったとき、`b` の先頭アドレスから `sizeof(DMI_barrier_t)` バイトをグローバルアドレス空間経由で他の DMI スレッドに渡したとき、渡された DMI スレッドは `b` を使うことができます。`b` はグローバル不透明オブジェクトなので意味が保存されるからです。要するに、ローカル不透明オブジェクトはその DMI スレッドの中でしか使えず、グローバル不透明オブジェクトは DMI スレッド間で受け渡しても使えるということです。

具体的には、グローバル構造体とローカル構造体には以下のようなものがあります：

```
グローバル構造体  DMI_thread_t ,DMI_node_t ,DMI_mutex_t ,DMI_cond_t ,DMI_rwlock_t
ローカル構造体   DMI_local_status_t ,DMI_local_barrier_t ,DMI_local_rwlock_t ,
                  DMI_local_group_t
```

グローバル不透明オブジェクトとローカル不透明オブジェクトを用いたプログラミング方法については、??で詳しく解説しています。



Chap.3 プロセスとスレッド

DMI では、計算規模を動的に拡張/縮小させることができますが、この計算規模の拡張/縮小は、プロセスを参加/脱退させたりプロセス上のスレッドを生成/回収させたりすることで実現します。本章では、どのようにしてプロセスを参加/脱退させるのか、スレッドを生成/回収するのかを説明した上で、DMI プログラミングによって計算規模を動的に拡張/縮小する方法を解説します。

1 基本事項

■ 1-1 DMI プロセスと DMI スレッド

DMI のシステム構成を Fig.?? に示します。DMI では、各ノードに任意の数の DMI プロセスを立ち上げることができます。また、各 DMI プロセスには任意の数の DMI スレッドを立ち上げることができます。DMI プログラムに記述されたコードを実際に行うのは DMI スレッドです。DMI プロセスや DMI スレッドは任意の数だけ立ち上げることができますが、性能を考慮した場合、1 個のノードあたり 1 個の DMI プロセスを、1 個の CPU コアあたり 1 個の DMI スレッドを立ち上げるのが理想的です。

■ 1-2 DMI プログラミングにおける実行モデル

DMI では、利用可能なノードが増えたときには並列計算にノードを新たに追加したり、利用可能なノードが減ったときには並列計算からノードを削除したりすることで、計算規模を動的に変化させるようなプログラムを記述できるようにしています。これは、MPI や多くの分散共有メモリ処理系などの既存の分散プログラミング処理系では実現できない機能であり、よって、これら既存の分散プログラミング処理系とは実行モデルが大きく異なります。MPI などの既存の多くの分散プログラミング処理系は SPMD 型の実行モデルを採用しているのに対して、DMI ではスレッド fork/join 型の実行モデルを採用しています。ここでは DMI の実行モデルについて解説します。

DMI の実行モデルは pthread プログラミングと似ています。DMI プログラムは以下の形式で記述します：

```
void DMI_main(int argc, char **argv)
{
    ...;
}

int64_t DMI_thread(int64_t addr)
{
    ...;
}
```

DMI_main(...) がプログラムの実行時に最初に 1 回だけ呼び出される関数です。DMI_thread(...) は DMI スレッドを起動したときに実行される関数です。プログラムの実行時に 1 回だけ DMI_main(...) が実行され、以降 DMI スレッドを生成すると DMI_thread(...) が実行されて並列処理が実現されるという部分は、pthread プログラミングと同じです。

ところが、DMI の場合には、最初から最後まで計算規模が一定であるようなプログラムだけではなく、プログラムを実行中に動的に DMI プロセスに参加/脱退させることで計算規模を変化させるようなプログラムも記述できるようにするため、単純な pthread プログラミングと比べると、プログラミングはやや複雑になっています。まず、DMI プログラミングにおいては、動的に DMI プロセスに参加/脱退させたり、その参加/脱退イベントをハンドリングしたりするための処理が必要です。また、DMI では各 DMI プロセス上に DMI スレッドを生成して並列処理を実現するわけなので、DMI プロセスが参加してきたときにはその DMI プロセス上に DMI スレッドを生成したり、DMI プロセスが脱退しようとしているときにはその DMI プロセスから DMI スレッドを回収したりするような処理も必要です。具体的に、DMI プロセスの参加/脱退や DMI スレッドの生成/回収をどう実現するかは次節以降で詳しく説明していきますが、おおよその流れは Fig.?? のようになります：

- (1) あるノードを計算に参加させたい場合、そのノード上に何らかの方法で DMI プロセスを生成して、DMI プログラムに対して参加通知を送ります。
- (2) DMI プログラム側で、その DMI プロセスの参加通知を許可して、その DMI プロセス上に複数の DMI スレッドを生成します。これによって計算規模が拡大します。
- (3) あるノードを計算から脱退させたい場合、そのノード上の DMI プロセスに何らかの方法で脱退通知を送ります。
- (4) DMI プログラム側で、その DMI プロセスの脱退通知を許可して、その DMI プロセス上の DMI スレッド全てを回収するなどして DMI プロセス上の DMI スレッドすべてを消したあとで、その DMI プロセスを脱退させます。これにより計算規模が縮小します。

2 プロセスの参加/脱退

■ 2-1 dmirun コマンド

2-1-1 参加通知の送り方

dmirun コマンドは、ノード上に DMI プロセスを生成して参加通知を送るための最も基本的なコマンドです。ノード i を参加させる場合、ノード i のコマンドラインで、

```
$ dmirun ./a.out [./a.out の引数]
```

のように dmirun コマンドを実行すると、ノード i 上に DMI プロセスが生成されて DMI_main(...) が呼び出されます。つまり、この dmirun コマンドは、何らかすでに実行されている DMI プログラムに対して参加通知を送るのではなく、新しい DMI プログラムを 1 個起動します。DMI プログラムを実行するために一番最初に実行するコマンドです。

一方、新たに DMI プログラムを実行するのではなく、すでに実行中の DMI プログラムに対して参加通知を送るためには、ノード i のコマンドラインで、

```
$ dmirun -i kototoi001 ./a.out
```

のように dmirun コマンドを実行すると、kototoi001 というホスト名のノードで実行されている DMI プロセスが属する DMI プログラムに対して参加通知が送られます。なお、参加/脱退通知は「特定の DMI プロセス」に対して送るものではなく、「実行中の DMI プログラム」に対して送るものです。よって、`-i` オプションに指定するホスト名としては、参加させたい DMI プログラムを実行中のどのノードのホスト名を指定してもかまいません。ここでホスト名を指定するのは、「このホスト名のノードが参加している DMI プログラムに参加したい」という意味であって、指定されたホスト名のノードが参加処理に関して特別何かの役割を担うわけではありません。このように `-i` オプション付きで dmirun コマンドを実行した場合、DMI_main(...) は実行されず、DMI プログラム側がノード i の参加通知を検知して許可し、ノード i 上に DMI スレッドを生成しない限り何も起きません。参加通知を送ったからといって自動的に DMI スレッドが生成されて並列処理に参加できるわけではないため、参加通知を検知して許可し、DMI スレッドを生成するように DMI プログラムを記述しておかなければならないという点に注意してください。参加通知を許可して DMI スレッドを生成するための方法は??節で説明します。やがて DMI スレッドが生成されると、ノード i 上で DMI_thread(...) が実行されます。なお、`-i` オプション付きで dmirun コマンドを実行した場合には DMI_main(...) が実行されないため、`./a.out` にコマンドライン引数を指定しても意味がありません。

2-1-2 脱退通知の送り方

ノード i 上の DMI プロセスを脱退させるためには、該当の DMI プロセスに対して SIGINT シグナルを送ります。最も単純な方法は、先ほど dmirun コマンドを実行させたコマンドラインに、直接 Ctrl+C を送る方法です：

```
$ dmirun -i kototoi001 ./a.out # 先ほど実行したコマンド
=== initialized ===
=== joined ===
```

```

started
started
started
=== opened ===
started
212
213
214
215
220
219
221
224
225                                # 処理が進行しているときに...
=== closed ===                    # Ctrl+C を入力する (SIGINT を送る)
226
finished
finished
finished
finished
=== left ===
=== finalized ===
$                                # 脱退が完了してコマンドラインが返ってくる

```

また、該当するプロセスを検索して `kill` コマンドで SIGINT シグナルを送る方法もあります：

```
$ ps aux | grep './a.out' | awk '{print $2}' | xargs kill -INT
```

SIGINT シグナルを送るとノード i の脱退通知が DMI プログラムに対して送られます。そして、DMI プログラム側でこの脱退通知を検知し、ノード i 上の DMI スレッドがすべて回収されてノード i の脱退を許可すると、ノード i の脱退が完了します。脱退通知を送ったからといって自動的に並列処理を脱退できるわけではなく、脱退通知を検知して、そのノード上の DMI スレッドをすべて回収したあとで脱退を許可するように DMI プログラムを記述しておかなければならないという点に注意してください。DMI スレッドを回収したあとで脱退通知を許可するための方法は??節で説明します。

2-1-3 進捗状況の表示

以上をまとめると、ノード i の参加処理の手順は以下のようになります：

- (1) `dmirun` コマンドを使い、すでに参加中の適当なノードを指定して参加通知を送ります。
- (2) DMI プログラムによってノード i の参加通知が検知されます。
- (3) DMI プログラムによって参加が許可されます。
- (4) DMI プログラムによってノード i 上に DMI スレッドが生成され、計算規模が拡張します。

一方で、ノード i の脱退処理の手順は以下のようになります：

- (1) SIGINT シグナルを送ることで、脱退通知を送ります。
- (2) DMI プログラムによってノード i の脱退通知が許可されます。
- (3) DMI プログラムによってノード i 上の DMI スレッドが回収され、計算規模が縮小します。

(4) DMI プログラムによって脱退が許可されます。

このような参加/脱退の進捗状況を，dmirun コマンドは以下のように表示します：

```
$ dmirun -i kototoi001 ./a.out
=== initialized ===      # DMI プロセスが生成される
=== joined ===          # 参加通知が送られる
=== opened ===          # 参加が許可される
...
...                      # DMI プログラムの出力
...
=== closed ===          # 脱退通知が送られる
=== left ===            # 脱退が許可される
=== finalized ===       # DMI プロセスが破棄される
$
```

たとえば，`=== closed ===`と出力されているにもかかわらず`=== left ===`と表示されない状態は，脱退通知はすでに送られているものの脱退がまだ許可されていない状態を意味します。

2-1-4 コマンドラインオプション

以上が dmirun コマンドの基本的な使い方ですが，dmirun コマンドには他にも多様なオプションがあります。

```
$ dmirun -h
```

と入力することで，各オプションの意味を調べることができます。ここでは，各オプションの意味を詳細に説明します：

- `-i` <文字列>オプション：

すでに実行中の DMI プログラムに参加するときに，どのノードが実行している DMI プログラムに参加するのかを指定します。`-i` オプションのあとには，ホスト名もしくは IP アドレスが指定できます。たとえば，

```
$ dmirun -i kototoi001 ./a.out
```

とすれば，kototoi001 が実行している DMI プログラムに参加通知を送ります。

- `-l` <数字>オプション：

同一ノードで複数の DMI プログラムが実行されている場合，ホスト名だけでは区別できないためポート番号で区別する必要があります。`-i` オプションは，dmirun コマンドで生成される DMI プロセスが使用するポート番号を明示的に指定できます。たとえば，

```
$ dmirun -i kototoi001 -l 12345 ./a.out
```

とすれば，この dmirun コマンドによって生成される DMI プロセスは 12345 番ポートに関連付けられます。`-l` オプションを省略すると，`-l 7880` が指定されたものと判断されます。

- `-p <数字>` オプション :

すでに実行中の DMI プログラムに参加するときに、どの DMI プログラムに参加するかをホスト名とポート番号で指示する場合のポート番号を指示します。よって、`-p` オプションは必ず `-i` オプションと同時に使用されます。つまり、`-i` オプションで指定されたホスト名で生成されている DMI プロセスのうち、どのポート番号に関連付けられている DMI プロセスが実行している DMI プログラムに参加するかを明示的に指定できます。たとえば、`kototoi001` のノード上で、

```
$ dmirun -i kototoi000 -l 12345 ./a.out
```

として生成された DMI プロセスと、

```
$ dmirun -i kototoi000 -l 67890 ./a.out
```

として生成された DMI プロセスがあったとします。このとき、`kototoi002` のノードを、`kototoi001` のノードで 12345 番ポートに関連付けられている DMI プロセスが実行している DMI プログラムに参加させる場合には、`kototoi002` のノードのコマンドラインで、

```
$ dmirun -i kototoi001 -p 12345 ./a.out
```

と指定します。以上をまとめると、ノード *i* のコマンドラインで、

```
$ dmirun -i iii -p ppp -l 111 ./a.out
```

と指定することの意味は、ノード *i* 上に 111 番ポートに関連付けられた DMI プロセスを生成した上で、`iii` というホスト名 (または IP アドレス) のノードで `xxx` 番ポートに関連付けられている DMI プロセスが実行している DMI プログラムに対して参加通知を送る、という意味です。`-p` オプションを省略すると、`-p 7880` が指定されたものと判断されます。

- `-s <数字>` オプション :

この DMI プロセスが DMI プログラムに対して提供するメモリ量を指定します。ここで指定するメモリ量は、各 DMI プロセスのメモリプールの使用上限値です。メモリプールに関しては??を参照してください。`-s` オプションを省略すると、メモリプールの使用上限値は 2GB に指定されます。

- `-h` オプション :

オプションの一覧との意味を表示します。

- `-v` オプション :

DMI のバージョンを表示します。

2-1-5 並列シェルを使って一括して参加/脱退させる

以上のように、`dmirun` コマンドを使うことで、DMI プログラムの実行中に任意の数の DMI プロセスを参加/脱退通知を送ることができますが、参加/脱退させるたびに該当ノードにログインして `dmirun` コマンドを入力するのは大変面倒です。これに対する 1 つの解決策は、SSH でコマンドを投入するような

シェルスクリプトを書くことですが、意図するタイミングで脱退できるように SIGINT シグナルを送れるようなシェルスクリプトを書くのは難しいです。このような場合、複数のノードに対して同時にコマンドを投入できるような並列シェルを使うと大変便利です。たとえば、GXP という並列シェルを使うと、意図した複数のノードの一括参加や一括脱退がインタラクティブに容易に実現できます。GXP に関しては、以下の Web サイトにドキュメントが揃っているのでご参照ください：

<http://www.logos.ic.i.u-tokyo.ac.jp/gxp/>

■ 2-2 dmimw コマンド

2-2-1 概要

このように `dmirun` コマンドは動的に計算規模を拡張/縮小させることに対する柔軟性は高いわけですが、最初から最後まで計算規模が一定であるようなアプリケーションを実行したい場合には、並列シェルを使ったとしても、1 ノードごとに参加通知を送り 1 ノードごとに脱退通知を送るのは面倒です。そこで DMI では、最初から最後まで計算規模が一定であるようなアプリケーションをより簡単に実行するために、`dmimw` コマンドを提供しています^{*1}。`dmimw` コマンドでは、ファイルにノードを羅列しておくとし、そのノードに対して一括して参加通知を送ったあと、一括して脱退通知を送ることができます。

2-2-2 参加通知の送り方

`dmimw` コマンドを使うには、まず以下のように 1 行に 1 個ずつホスト名を記述したノードファイルを用意します。ここでは `nodefile.txt` という名前で保存するとします：

```
kototoi000
kototoi001
kototoi002
kototoi003
kototoi004
kototoi005
kototoi006
kototoi007
kototoi008
kototoi009
kototoi010
kototoi011
kototoi012
```

次に、適当なノードのコマンドラインから、

```
$ dmimw -f nodefile.txt -n 8 ./a.out [./a.out の引数]
```

のように `dmimw` コマンドを実行すると、ノードファイルの一番上に書いてあるノードで DMI プログラムが新たに起動されたあと、上から 2 番目から 8 番目までの合計 7 個のノードがこの DMI プログラムに

^{*1} `dmimw` の `mw` は master worker の略です。

対して参加通知を送ります。つまり、合計 8 個のノードが参加します。具体的には、nodefile.txt の一番上に書いてある kototoi000 のノードで、

```
$ dmirun ./a.out [./a.out の引数]
```

というコマンドが実行されて DMI プログラムが新たに起動されたあと、kototoi001、kototoi002、…、kototoi007 の 7 個のノードで、

```
$ dmirun -i kototoi000 ./a.out
```

が実行されて、この DMI プログラムに対して参加通知が送られます。dmimw コマンドは各ノードに 1 個ずつログインして dmirun コマンドを入力する手間を省くだけであり、dmimw コマンドを使う場合であっても、ファイルで指定したノード上に自動的に DMI スレッドが生成されて並列処理が行われるわけではなく、参加通知を検知して許可し、DMI スレッドを生成するように DMI プログラムを記述しておかなければならないという点に注意してください。

なお、dmimw コマンドを実行するためには、この dmimw コマンドを実行するノードから、DMI プロセスが生成される各ノードに対して SSH 接続できる必要があります。

2-2-3 脱退通知の送り方

DMI プログラムを終了する際には、dmimw コマンドに対して SIGINT シグナルを送ります。具体的には、dmimw コマンドのコマンドラインに対して Ctrl+C を送るか、または kill コマンドで SIGINT シグナルを送ります。これにより、参加しているすべてのノードに対して脱退通知を送ることができます。上の例の場合、8 個のノードに対して脱退通知が送られます。dmimw コマンドは各ノードに 1 個ずつログインして SIGINT シグナルを送る手間を省くだけであり、dmimw コマンドを使う場合であっても、SIGINT シグナルを送っただけですべてのノードが自動的に並列処理を脱退するわけではなく、脱退通知を検知して、そのノード上の DMI スレッドをすべて回収したあとで脱退を許可するように DMI プログラムを記述しておかなければならないという点に注意してください。

2-2-4 ノードファイルのフォーマット

ノードファイルには、各ノードのホスト名または IP アドレスのあとに、そのノード上で dmirun コマンドが実行されるときに付けたいオプションを付記することができます。たとえば、

```
kototoi000 -s 1024000000
kototoi001 -s 1024000000
kototoi002 -s 1024000000
kototoi003 -s 1024000000
kototoi003 -l 12345
kototoi004
kototoi005
kototoi006
kototoi007
kototoi008
kototoi009
kototoi010
kototoi011
```

```
kototoi012
```

のように記述した上で、

```
$ dmimw -f nodefile.txt -n 8 ./a.out [./a.out の引数]
```

として dmimw コマンドを実行すると、kototoi000 では、

```
$ dmirun -s 1024000000 ./a.out [./a.out の引数]
```

というコマンドが実行され、kototoi001 から kototoi003 までのノードでは、

```
$ dmirun -i kototoi000 -s 1024000000 ./a.out
```

というコマンドが実行され、さらに kototoi003 では、

```
$ dmirun -i kototoi000 -l 12345 ./a.out
```

というコマンドが実行され^{*2}、kototoi004 から kototoi006 までのノードでは、

```
$ dmirun -i kototoi000 ./a.out
```

というコマンドが実行されます。

2-2-5 コマンドラインオプション

dmimw コマンドに指定できるオプションは以下のとおりです：

- **-f <文字列>オプション：**
ノードファイルを指定します。
- **-n <数字>オプション：**
ノードファイルのうち、上から何個のノードを DMI プログラムに参加させるかを指定します。-n オプションを省略した場合、ノードファイルに記述されたすべてのノードが DMI プログラムに参加します。
- **-m <数字>オプション：**
一番最初に新たに DMI プログラムが起動されるノードを、ノードファイルの上から何行目のノードにするかを指定します。-m オプションを省略した場合、-m 1 が指定されたものと判断されます。つまり、デフォルトではノードファイルの 1 行目に記述されたノードで新たに DMI プログラムが起動されます。
- **-h オプション：**
dmimw コマンドのオプションの一覧と意味を表示します。

^{*2} 1 個のノード上では 1 個のポート番号には 1 個の DMI プロセスしか関連付けられないため、1 個のノード上に複数の DMI プロセスを生成したい場合には、-l オプションを付けることで明示的にポート番号を区別しないとエラーになります。

■ 2-3 参加/脱退通知の検知と許可

2-3-1 概要

繰り返しますが、`dmirun` コマンドや `dmimw` コマンドは単に参加/脱退通知を送るだけです。実際に並列処理に参加するためには、DMI プログラム側で参加通知を検知して参加を許可し、DMI スレッドを生成する必要があります。また、実際に並列処理から脱退するためには、DMI プログラム側で脱退通知を検知して、そのノード上のすべての DMI スレッドを回収したあとで脱退を許可する必要があります。よって、ここでは参加/脱退通知を検知して許可するための手順を解説します。

2-3-2 参加/脱退通知を検知する API

参加/脱退通知を検知するには、`DMI_poll(...)` または `DMI_peek(...)` を使います。正確な API は以下のとおりです：

- `int32_t DMI_poll(DMI_node_t *node_ptr);`
参加/脱退通知が来るまで待機し、参加/脱退通知が生じた場合に、その参加/脱退通知を送った DMI プロセスに関するさまざまな情報を `node_ptr` に格納して返ります。`DMI_node_t` の構造体のメンバについては後述します。参加/脱退通知が生じない限り、`DMI_poll(...)` は返りません。
- `int32_t DMI_peek(DMI_node_t *node_ptr, int32_t *flag_ptr);`
`DMI_peek(...)` が呼び出された時点で参加/脱退通知が存在しているかどうか調べ、存在しているならば、`flag_ptr` の値を `DMI_TRUE` にした上で、その参加/脱退通知を送った DMI プロセスに関するさまざまな情報を `node_ptr` に格納して返ります。存在していないならば、`flag_ptr` の値を `DMI_FALSE` にした上で返ります。つまり、`DMI_poll(...)` は参加/脱退通知が存在するまで待機するのに対して、`DMI_peek(...)` は存在しなくてもすぐに返ります。

各 DMI プロセスが送った DMI プログラムに対して生じた参加/脱退通知は、グローバルで FIFO なキューに保存されています。そして、`DMI_poll(...)` または `DMI_peek(...)` を 1 回呼び出すたびに、キューから 1 個ずつ参加/脱退通知が取り出され、その参加/脱退通知を送った DMI プロセスの情報を返します。よって、複数の参加/脱退通知をほぼ同時に送ったとしてもそれが失われることはありません。また、複数の DMI プロセスがほぼ同時に `DMI_poll(...)` や `DMI_peek(...)` を呼び出したとしても、1 個の参加/脱退通知が複数の `DMI_poll(...)` や `DMI_peek(...)` に検知されることもありません。

2-3-3 DMI_node_t 構造体

`DMI_node_t` 構造体はグローバル不透明オブジェクトであり、参加/脱退通知を送ってきた DMI プロセスに関するさまざまな情報を含んでおり、DMI プロセス上に DMI スレッドを生成する場合などにこの情報を利用することができます：

```
typedef struct DMI_node_t
{
    int32_t state;
    int32_t dmi_id;
```



```
int32_t core;
int64_t memory;
char hostname[IP_SIZE];
}DMI_node_t;
```

これらのメンバの意味は次のとおりです：

- `state`：参加通知の場合には `DMI_OPEN`，脱退の場合には `DMI_CLOSE` の値になっています．
- `dmi_id`：参加/脱退通知を送ってきた DMI プロセスに対して割り当てられた番号です．この番号のことをその DMI プロセスの `id` と呼びます．各 DMI プロセスの `id` は，その DMI プロセスが生成されてから消滅するまで不変であり，かつその時点で存在している他のどの DMI プロセスの `id` とも異なることが保証されています．よって，ある DMI プロセスが参加通知を送ってきた際の `id` と脱退通知を送ってきた際の `id` は当然一致します．ただし，DMI プログラムの実行を通じて，異なる DMI プロセスに対して同一の `id` が割り振られる可能性はあります．たとえば，時刻 0 から時刻 10 までに存在した DMI プロセスと，時刻 20 から時刻 30 までに存在した DMI プロセスに同一の `id` が割り振られることがあります．
- `core`：参加/脱退通知を送ってきた DMI プロセスが所属するノードの CPU コア数です．性能上は，CPU コア数の数だけ DMI スレッドを生成することが望ましいため，DMI スレッド生成時にこの値を参照する場合があります．
- `memory`：参加/脱退通知を送ってきた DMI プロセスが提供しているメモリプールのサイズです．メモリプールに関しては??を参照してください．
- `hostname`：参加/脱退通知を送ってきた DMI プロセスが所属するノードのホスト名です．

このように，`DMI_poll(...)` または `DMI_peek(...)` を使うことで，参加/脱退通知を送ってきた DMI プロセスに関するさまざまな情報を取得することができます．

2-3-4 参加/脱退を許可する API

参加通知を検知したあと，参加を許可する必要があります．参加を許可する API は以下のとおりです：

- `int32_t DMI_welcome(int32_t dmi_id);` :
id が `dmi_id` の DMI プロセスの参加を許可します．

`DMI_welcome(...)` によって参加が許可されると，参加通知を送った DMI プロセスの `dmirun` コマンドのプログレス表示において `=== opened ===` が表示されます．

一方で，脱退通知を検知したとき，そのノード上のすべての DMI スレッドを回収するなどしたあと，脱退を許可する必要があります．脱退を許可する API は以下のとおりです：

- `int32_t DMI_goodbye(int32_t dmi_id);` :
id が `dmi_id` の DMI プロセスの脱退を許可します．

`DMI_goodbye(...)` によって脱退が許可されると，脱退通知を送った DMI プロセスにおいて，`=== left ===` が表示されます．

このように、DMI では参加/脱退を検知してから参加/脱退を許可するというステップが必要であり、一見これは単に面倒なだけに思えます。しかし、いざ並列アプリケーションを記述しようとする、仮に外部からのコマンドによって任意のタイミングで参加/脱退が行われてしまうようでは、安全な並列アプリケーションを記述するのが非常に難しくなります。当然ですが、並列アプリケーションにおいて参加/脱退が行われても良いタイミングは任意ではなく決まっています。よって、意図するタイミングで DMI プログラム側から参加/脱退を指示できるようなプログラミングインタフェースになっていないと、動的な参加/脱退に対応した DMI プログラムを記述できません。このような事情を考慮した結果、現在のような API になっています。

2-3-5 具体例

ここまで順を追って説明してきた DMI プロセスの参加/脱退の手順を、具体例を通じてまとめます。以下のプログラムは、動的な DMI プロセスの参加/脱退を処理します。ただし、話を簡単化するために DMI スレッドの生成/破棄に関する処理を省いているため、計算規模は何も変化しません：

```
#include "dmi_api.h"

void DMI_main(int argc, char **argv)
{
    DMI_node_t node;
    int my_dmi_id;

    DMI_rank(&my_dmi_id); /* この DMI プロセスの id を取得 */
    while(1)
    {
        DMI_poll(&node); /* 参加/脱退通知を待機 */
        if(node.state == DMI_OPEN) /* 参加通知を検知したら */
        {
            printf("welcome %d!", node.dmi_id); /* 参加通知を送ってきた DMI プロセスの id を出力 */
            DMI_welcome(node.dmi_id); /* 参加を許可 */
        }
        else if(node.state == DMI_CLOSE) /* 脱退通知を検知したら */
        {
            printf("goodbye %d!", node.dmi_id); /* 脱退通知を送ってきた DMI プロセスの id を出力 */
            DMI_goodbye(node.dmi_id); /* 脱退を許可 */
            if(node.dmi_id == my_dmi_id) /* その脱退通知が自分の DMI プロセスに対するものだったら */
            {
                break; /* 無限ループを抜けて DMI プログラムを終了 */
            }
        }
    }
    return;
}
```

上記のプログラムを `ex.c` という名前で保存して、`kototoi000` というノードで以下のコマンドを入力して DMI プログラムを起動します：

```
$ dmicc -O3 ex.c # コンパイル
$ dmirun ./a.out # 実行
```

その後, kototoi001, kototoi002, kototoi003 の各ノードで以下のコマンドを入力し, 今実行した DMI プログラムに参加させます:

```
$ dmirun -i kototoi000 ./a.out
```

すると, 以下のような出力が各ノードのコマンドラインに得られ, 各ノード上の DMI プロセスの参加が完了したことがわかります:

```
$ dmirun -i kototoi000 ./a.out
=== initialized ===
=== joined ===
=== opened ===
```

また, このとき参加/脱退の検知と許可を行っている kototoi000 では,

```
$ dmirun ./a.out
=== initialized ===
=== joined ===
welcome 0!
=== opened ===
welcome 1!
welcome 2!
welcome 3!
```

のような出力が得られており, 確かに kototoi001, kototoi002, kototoi003 の DMI プロセスに対して参加を許可したことがわかります。

続いて, kototoi001, kototoi002, kototoi003 の各ノードで Ctrl+C を入力し, 3 つの DMI プロセスを DMI プログラムから脱退させます:

```
$ dmirun -i kototoi000 ./a.out
=== initialized ===
=== joined ===
=== opened ===
# ここで Ctrl+C を入力
```

すると, 以下のような出力が各ノードのコマンドラインに得られ, 各ノード上の DMI プロセスの脱退が完了したことがわかります:

```
$ dmirun -i kototoi000 ./a.out
=== initialized ===
=== joined ===
=== opened ===
=== closed ===
=== left ===
=== finalized ===
```

また, kototoi001, kototoi002, kototoi003 の各ノードで Ctrl+C を入力した後で, kototoi000 のコマ

ンドラインを見ると，

```
$ dmirun ./a.out
=== initialized ===
=== joined ===
welcome 0!
=== opened ===
welcome 1!
welcome 2!
welcome 3!
goodbye 3!
goodbye 2!
goodbye 1!
$
```

と表示されており，確かに kototoi001，kototoi002，kototoi003 の脱退を許可したことがわかります．最後に，kototoi000 で Ctrl+C を入力すると，DMI プログラムが終了します：

```
$ dmirun ./a.out
=== initialized ===
=== joined ===
welcome 0!
=== opened ===
welcome 1!
welcome 2!
welcome 3!
goodbye 3!
goodbye 2!
goodbye 1!
=== closed ===
goodbye 0!
=== left ===
=== finalized ===
$
```

なお，上記の手順の中で，繰り返し kototoi001，kototoi002，kototoi003 を参加/脱退させたり，参加時に指定するノードとして kototoi000 以外の参加ノードを指定したりすることも可能です．また，上記の DMI プログラムは，DMI_main(...) を実行している DMI プロセス自身に対する脱退通知があった場合に，DMI プログラムを終了する仕様にしてあるため，DMI_main(...) を実行している DMI プロセスに脱退通知を送った時点で，他に DMI プロセスがまだ実行中だと，それらの DMI プロセスの挙動は未定義になることに注意してください．

2-3-6 その他の API

プロセスの参加/脱退に関連して，以下のような API があります：

- `int32_t DMI_rank(int32_t *dmi_id_ptr);` :

この DMI_rank(...) を呼び出した DMI プロセスの id を dmi_id_ptr に格納します．

- `int32_t DMI_nodes(DMI_node_t *node_array, int32_t *num_ptr, int32_t capacity);` :

この `DMI_nodes(...)` を呼び出した時点で参加している DMI プロセスの情報を、最大 `capacity` 個だけ配列 `node_array` に格納します。`num_ptr` には、実際に配列 `node_array` に格納した DMI プロセス情報の個数が格納されます。この API によって、参加中のすべての DMI プロセスの情報を一括して取得できます。

3 スレッドの生成/回収

■ 3-1 スレッドの生成/回収/detach

計算規模を拡張するためには参加を許可したあとで DMI スレッドを生成する必要があり、計算規模を縮小するためには DMI スレッドを回収する必要があります。DMI では、`pthread` プログラミングとの対応性を重視した API を提供しており、`pthread` プログラミングと類似した API でスレッドの生成/回収/detach を行えます：

- `int32_t DMI_create(DMI_thread_t *thread_ptr, int32_t dmi_id, int64_t addr, DMI_local_status_t *status);`
`dmi_id` の `id` を持つ DMI プロセス上に DMI スレッドを生成します。DMI スレッドは、`id` が `dmi_id` の DMI プロセス上で `int64_t DMI_thread(int64_t addr) { ... }` という関数として実行され始めます。`DMI_create(...)` に渡した `addr` の値はそのまま `DMI_thread(...)` の引数である `addr` に渡されます。つまり、この `addr` が DMI スレッド起動時に与えられる引数です。メモリアクセスについては??節で詳しく説明しますが、DMI スレッドに渡したいデータを適当なグローバルメモリに格納しておき、そのグローバルメモリアドレスを `addr` として渡すことで、任意のサイズのデータを DMI スレッドに教えることができます。生成した DMI スレッドのハンドルが `thread_ptr` に格納されます。このハンドルはグローバル不透明オブジェクトです。`status` を指定することで非同期操作にできます。
- `int32_t DMI_join(DMI_thread_t thread, int64_t *addr_ptr, DMI_local_status_t *status);`
`thread` のハンドルで示される DMI スレッドを回収します。この `DMI_join(...)` は、`int64_t DMI_thread(int64_t addr) { ... }` が終了したときに返り、`DMI_thread(...)` の返り値が `addr_ptr` に格納されます。`status` を指定することで非同期操作にできます。
- `int32_t DMI_detach(DMI_thread_t thread, DMI_local_status_t *status);`
`thread` のハンドルで示される DMI スレッドを detach します。この `DMI_detach(...)` はすぐに返ります。いったん detach を行くと以降では DMI スレッドを制御できなくなるため、`int64_t DMI_thread(int64_t addr) { ... }` の完了を待ち合わせるなどとはできなくなります。`status` を指定することで非同期操作にできます。
- `int32_t DMI_self(DMI_thread_t *thread_ptr);`
この `DMI_self(...)` を呼び出した DMI スレッドのハンドルを `thread_ptr` に返します。

■ 3-2 具体例

??節で説明した DMI プログラムを拡張して、参加してきた DMI プロセスに対して DMI スレッドを生成し、脱退する DMI プロセスから DMI スレッドを回収する処理を加えると以下のようになります。このプログラムでは、各 DMI スレッドにランクを割り当て、各 DMI スレッドに表示させています。グローバルアドレス空間に対するメモリ確保/解放や read/write の API についての詳細は??を参照してください：

```
#include "dmi_api.h"

#define NODE_MAX 64 /* 簡単のため DMI プロセス数の上限値を決め打つ */
#define CORE_MAX 8 /* 簡単のため各ノードに生成する DMI スレッド数の上限値を決め打つ */
#define THREAD_MAX (NODE_MAX * CORE_MAX) /* 生成される DMI スレッド数の上限値 */

typedef struct targ_t
{
    int32_t rank; /* DMI スレッドのランク */
    int32_t flag; /* DMI スレッドに対して終了を指示するために使う */
}targ_t;

void DMI_main(int argc, char **argv)
{
    targ_t targ;
    DMI_node_t node;
    DMI_thread_t threads[THREAD_MAX];
    int i, my_dmi_id, rank, flag;
    int64_t targ_addr, my_targ_addr;

    DMI_rank(&my_dmi_id); /* この DMI プロセスの id を取得 */
    DMI_mmap(&targ_addr, sizeof(targ_t), THREAD_MAX, NULL); /* 各 DMI スレッドに渡す引数のためのグ
グローバルメモリを確保 */
    for(rank = 0; rank < THREAD_MAX; rank++) /* 各 DMI スレッドに渡す引数を初期化 */
    {
        targ.rank = rank;
        targ.flag = 0;
        DMI_write(targ_addr + rank * sizeof(targ_t), sizeof(targ_t), &targ, DMI_EXCLUSIVE_WRITE, NU
グローバルアドレス空間に書き込む */
    }
    while(1)
    {
        DMI_poll(&node); /* 参加/脱退通知を待機 */
        if(node.state == DMI_OPEN) /* 参加通知を検知したら */
        {
            printf("welcome %d!\n", node.dmi_id);
            DMI_welcome(node.dmi_id); /* 参加を許可 */
            for(i = 0; i < node.core; i++) /* コア数だけ DMI スレッドを生成 */
            {
                rank = node.dmi_id * CORE_MAX + i; /* 生成する DMI スレッドのランクを決める */
                my_targ_addr = targ_addr + rank * sizeof(targ_t); /* 生成する DMI スレッドに渡す
引数が格納されているグローバルアドレスアドレスを求める */
                flag = 1; /* flag==1 である限り、DMI スレッドは走り続けていて良い */
                DMI_write((int64_t)&(((targ_t*)my_targ_addr)->flag), sizeof(int32_t), &flag, DMI_PU
の値をグローバルアドレス空間に書き込む */
                DMI_create(&threads[rank], node.dmi_id, my_targ_addr, NULL); /* DMI スレッド
```

```

を生成 */
    }
}
else if(node.state == DMI_CLOSE) /* 脱退通知を検知したら */
{
    for(i = 0; i < node.core; i++) /* その DMI プロセス上のすべての DMI スレッドに終了を指示 */
    {
        rank = node.dmi_id * CORE_MAX + i; /* DMI スレッドのランクを求める */
        my_targ_addr = targ_addr + rank * sizeof(targ_t);
        flag = 0; /* flag==0 として, DMI スレッドに終了を指示 */
        DMI_write((int64_t)&(((targ_t*)my_targ_addr)->flag), sizeof(int32_t), &flag, DMI_UPDATE_READ, N
の値をグローバルアドレス空間に書き込む */
    }
    for(i = 0; i < node.core; i++) /* その DMI プロセス上のすべての DMI スレッドを回収 */
    {
        rank = node.dmi_id * CORE_MAX + i;
        DMI_join(threads[rank], NULL, NULL); /* DMI スレッドを回収 */
    }
    printf("goodbye %d!\n", node.dmi_id);
    DMI_goodbye(node.dmi_id); /* 脱退を許可 */
    if(node.dmi_id == my_dmi_id) /* その脱退通知が自分の DMI プロセスに対するものだったら */
    {
        break; /* 無限ループを抜けて DMI プログラムを終了 */
    }
}
}
DMI_munmap(targ_addr, NULL); /* グローバルメモリを解放 */
return;
}

/* 各 DMI スレッドが実行する関数 */
int64_t DMI_thread(int64_t targ_addr)
{
    targ_t targ;
    int flag;

    DMI_read(targ_addr, sizeof(targ_t), &targ, DMI_GET_READ, NULL); /* この DMI スレッドに渡され
た引数を読み出す */
    printf("started! %d\n", targ.rank);
    while(1)
    {
        DMI_read((int64_t)&(((targ_t*)targ_addr)->flag), sizeof(int32_t), &flag, DMI_UPDATE_READ, N
の値を読む */
        if(flag == 0) /* flag が 0 だったら */
        {
            break; /* DMI スレッドを終了 */
        }
        printf("running! %d\n", targ.rank);
        sleep(1);
    }
    printf("finished! %d\n", targ.rank);
    return DMI_NULL;
}

```

上記のプログラムでは、参加通知を送ってきた DMI プロセスに対して DMI スレッドを生成し、脱退通知を送ってきた DMI プロセスから DMI スレッドを回収することで計算規模を増減させています。DMI スレッドを回収する部分で、`flag` を使っている部分が鍵です。DMI スレッドを回収するとはいえ、何もしなければ、その DMI スレッドは走り続けたままなので回収できません。そこで、`flag` が 1 ならば DMI スレッドは走り続けても良く、`flag` が 0 ならば DMI スレッドは終了しなければならないという意味で `flag` を使います。各 DMI スレッドは、定期的に `flag` の値を観察し 0 になっていたら終了するようにしておき、`DMI_main(...)` 側では回収したい DMI スレッドの `flag` を 0 に書き換えるようにすれば、DMI スレッドを終了させ回収することができるわけです。

実行方法は??節で述べた手順と同様です。DMI プロセスを参加/脱退させて DMI スレッドが生成/回収されるときの様子を観察してみてください。描く DMI スレッドが生成されると `started!`、消滅すると `finished!` が表示されます。

■ 3-3 補足

3-3-1 `DMI_main(...)` を途中で脱退させることは可能か

??で説明した DMI プログラムでは、一番最初に実行されるのは `DMI_main(...)` であり、一番最後まで実行されているのも `DMI_main(...)` です。つまり、`DMI_main(...)` は、DMI プログラムの実行が始まってから終わるまで常に実行されていることになります。これは、`DMI_main(...)` を実行している DMI プロセス、つまり、最初に DMI プログラムを起動した DMI プロセスは、DMI プログラムが終了するまでは脱退できないことを意味します。

ところが、実は DMI では、DMI プログラムが終了する前に `DMI_main(...)` を終了させて、最初に DMI プログラムを起動した DMI プロセスを脱退させることも可能です。この場合、`DMI_main(...)` を終了してしまった後は、各 DMI スレッドが実行している `DMI_thread(...)` たちだけが協調して動作することになります。このようなプログラミング技法を応用すると、DMI プログラムの実行開始から実行終了まで存在するような固定的なノードを設置することなく、Fig.??に示すように、計算環境を移動して渡り歩いていくような不思議な DMI プログラムを記述することも可能です。しかし、計算環境を渡り歩かせることは確かに DMI の処理系としては可能なのですが、このように固定的な DMI プロセスを一切設けることなく計算環境を渡り歩いていく並列プログラムを記述することは相当に難しいことです。この難しさは、DMI における参加/脱退の仕組みの難しさに起因しているのではなく、固定的な DMI プロセスを設けることなく計算環境を渡り歩いていけるような並列アルゴリズムを設計することの難しさに起因しています。



Chap.4 メモリアクセス

グローバルメモリアドレス処理系において，最も基本になるのは，グローバルメモリアドレス空間に対する read/write です．本章では，DMI におけるグローバルメモリアドレス空間の構成について基本的な概念を説明したあと，最も基本となる read/write の API について詳しく解説します．特に，選択的キャッシュ read/write は，アプリケーションを高速化するうえで非常に重要な手段です．そのあと，離散的な read/write，read-write-set など，より高度な read/write を実現する方法について解説します．

1 基本事項

■ 1-1 ローカルメモリとグローバルメモリ

DMI のシステム構成は，Fig.??に示すようになっており，各 DMI スレッドにとってローカルなアドレス空間と，全 DMI スレッドによって共有されているアドレス空間は明確に区別されています．DMI では，各 DMI スレッドにとってローカルなアドレス空間のことをローカルメモリアドレス空間，全 DMI スレッドによって共有されているメモリグローバルメモリアドレス空間と呼びます．また，ローカルメモリアドレス空間上のアドレスをローカルメモリアドレス，グローバルメモリアドレス空間上のアドレスをグローバルメモリアドレスと呼びます．当然，メモリアクセスを行うためには，アドレス空間上にメモリを確保/解放したり，そのメモリに read/write する必要がありますので，本章では，これらのアドレス空間に対するメモリの確保/解放，read/write をどのように行うかについて解説していきます．

DMI では，ローカルメモリアドレス空間に対しては，通常の `malloc(...)/free(...)/mmap(...)/munmap(...)/alloca(...)/brk(...)/sbrk(...)` などによって，メモリを確保/解放することができます．説明の都合上，1 回の `malloc(...)/mmap(...)` などによって確保されたメモリのことを，以降ではローカルメモリと呼びます．つまり，ローカルメモリとは C 言語で普通に確保/解放できる通常のメモリのことです．これらのメモリに対する read/write は，通常の変数参照や配列アクセスなどで行うことができます．

一方で，グローバルメモリアドレス空間に対しては，`DMI_mmap(...)/DMI_munmap(...)` によってメモリを確保/解放します．1 回の `DMI_mmap(...)` で確保されたメモリのことをグローバルメモ

りと呼びます。グローバルメモリアドレスは 64 ビット整数で表されます。また、ローカルメモリにおける NULL に対応する特殊なアドレスとして `DMI_NULL` が定義されています。グローバルメモリに対する `read/write` は、最も基本的には、`DMI_read(...)/DMI_write(...)` によって行います。この `DMI_read(...)/DMI_write(...)` は、おおよそ以下のような API になっています：

- `DMI_read(int64_t addr, int64_t size, void *buf, ...)`：
グローバルメモリアドレス `addr` から `size` バイトをローカルメモリアドレス `buf` に読み込む。
- `DMI_write(int64_t addr, int64_t size, void *buf, ...)`：
ローカルメモリアドレス `buf` から `size` バイトをグローバルメモリアドレス `addr` に書き込む。

多くの分散共有メモリでは、何かのアノテーションを付けてメモリを確保することでグローバルメモリが確保でき、あとは通常の変数参照や配列アクセスと同様のシンタックスでグローバルメモリに `read/write` することができます。しかし、DMI では、そのようなことはできず、グローバルメモリに `read/write` するためには、その都度 `DMI_read(...)/DMI_write(...)` を呼ぶ必要があります。このシンタックスは、プログラミングを相当に面倒にしますし、既存の `pthread` プログラムを DMI プログラムに移植しようと思った場合の変更作業も多くなるという欠点があります。しかし、プログラミングが「概念的」に難しいわけではなく、既存の `pthread` プログラムからの移植に関しても、変更すべき部分が「作業的」に増えるだけであって、概念的には `pthread` プログラムとの対応性を確保しています。さて、なぜこのような API 呼び出しの形式を採用しているかということ、それは性能のためです。DMI では、このように API 呼び出しの形式にすることで、グローバルメモリに対する各 `DMI_read(...)/DMI_write(...)` のたびに、API の引数としてさまざまな情報を与えられるようにし、各 `DMI_read(...)/DMI_write(...)` の挙動を細粒度かつ明示的にチューニングできるようにしています。多くの分散共有メモリ処理系や PGAS 処理系では、明示的なチューニングの行いにくさが問題になっていますが、DMI では、API 呼び出し形式を採用することで非常に柔軟なチューニング手段を提供することができます。具体的なチューニング手段については本章で解説していきます。

■ 1-2 プログラミングの基本指針

多くの分散共有メモリ処理系では、「グローバルメモリに対する 1 回の `read/write` をいかに高速に処理するか」を追求しているものが多いです。コンシステンシモデルの緩和や、メッセージ数の徹底的な削減、通信レイテンシの隠蔽など、実装上の多様な工夫が研究されています。これに対して DMI では、「1 回の `read/write` はちょっとくらい遅くても良い。その代わりに、1 回の `read/write` で高度なことを実現できるようにする」ことを基本思想としています。上記のように、API 呼び出しの形式をとっている時点で、DMI における 1 回の `read/write` が遅いのは容易に予想が付くと思います。つまり、グローバルメモリへの `read/write` のたびに関数呼び出しのオーバーヘッドがかかることになり、ローカルメモリとグローバルメモリを明確に区別しているがために、グローバルメモリへの `read/write` のたびに、ローカルメモリとグローバルメモリの間でのメモリコピーが必要になります。ただし、DMI では、本章で解説するように、非常に高度で多様な `read/write` のチューニング手段を提供しており、1 回の `read/write` で非常に充実したことを実現できます。多くのアプリケーション、特に分散処理が威力を発揮するようなある程度スケラブルなアプリケーションでは、DMI が提供するチューニング手段を上手に組み合わせることで、1

回の read/write の遅さを補うくらいの効果が得られる場合が多いと考えられます。

したがって、DMI で高性能なアプリケーションを記述する場合には、DMI の高度なチューニング手段をうまく組み合わせることで、`DMI_read(...)/DMI_write(...)` などの API 呼び出し回数を最小限に抑えることが鍵になります。DMI でアプリケーションを記述する際には、最小回数の `DMI_read(...)/DMI_write(...)` しか行わないようなプログラミングを意識してください。

2 メモリの確保/解放

■ 2-1 ページ

メモリ確保/解放の API の説明の前に、ページについて説明します。ページとは、DMI がコンシステンスを維持するうえでの単位のことです。DMI がグローバルメモリを確保すると、このグローバルメモリ全体はページ単位に分割されて管理されます。これらの各ページの実体は、各 DMI プロセスが持つメモリプールのどれかに確保されます。以降、`DMI_read(...)/DMI_write(...)` によってグローバルメモリに read/write すると、各 DMI プロセスのメモリプールをメモリ資源として、ページを単位としたキャッシュ管理が行われます。

たとえば、Fig.?? に示すように、DMI プロセス i のメモリプールだけに実体が存在するようなページ p に含まれる領域を、DMI プロセス j が `DMI_read(...)` したとすると、DMI プロセス j のメモリプールにはページ p が存在しないためページフォルトが発生します。そして、DMI が適切なコンシステンス維持を行ってページフォルトが解決され、やがて DMI プロセス j のメモリプールにページ p がキャッシュされます。次回以降、DMI プロセス j がページ p を `DMI_read(...)` する場合には、DMI プロセス j はすでにページ p をメモリプールに持っているため、この `DMI_read(...)` はページフォルトを発生させることなく、ローカルに高速に完了します。DMI における実際のキャッシュ機構は複雑でその正確な仕組みについては??で解説しますが、要するに、DMI では、各 DMI プロセスのメモリプールをメモリ資源として、ページを単位としたキャッシュ管理が行われるということです。ページは、マルチプロセッサマシンにおけるキャッシュラインに相当するものです。

■ 2-2 API

DMI では、`DMI_mmap(...)/DMI_munmap(...)` によってメモリを確保/解放します。正確な API は以下のとおりです：

- `int32_t DMI_mmap(int64_t *dmi_addr_ptr, int64_t page_size, int64_t page_num, DMI_local_status_t *status);`
グローバルメモリアドレス空間に、ページサイズが `page_size` のページを `page_num` 個確保し、そのグローバルメモリアドレスを `dmi_addr_ptr` に格納します。`status` を利用することで非同期呼び出しにできます。
- `int32_t DMI_munmap(int64_t dmi_addr, DMI_local_status_t *status);`
アドレスが `dmi_addr` のグローバルメモリを解放します。`status` を利用することで非同期呼び出

しにできます。

`DMI_mmap(...)` によって、アプリケーションの特性に合致した任意のページサイズを指定してグローバルメモリを確保することができます。高性能なプログラムを開発するためには、ページサイズは大きすぎても小さすぎてもいけません。これは一般のキャッシュに対していえることですが、ページサイズが大きすぎると false sharing が多発して性能が落ちますし、かといってページサイズが小さすぎるとページ管理のオーバーヘッドが大きくなります。ページサイズを小さくすると、DMI が管理する対象のページが増えてページ管理のためのオーバーヘッドが増えるのは当然ですが、それより重大なのはネットワーク通信時のバンド幅です。一般に、ある程度大きなサイズのデータを送る場合に、これを複数回に分けて送ると、1回でまとめて送る場合と比較してバンド幅が有効利用できず転送効率が落ちてしまうため、ネットワーク通信は可能な限り大きな単位で行われることが重要です。よって、DMI においては、ページサイズは、false sharing による悪影響が及ばない範囲で可能な限り大きくとるのが理想です。以上のことを一言でまとめると、アプリケーションの挙動を分析したうえで、ページフォルトの回数を最小化するようなページサイズを採用するのが望ましい、ということになります。

たとえば、??節で紹介した、行列行列積を横ブロック分割で行うプログラムを考えてみます。このプログラムでは、行列 *A* と行列 *C* は DMI スレッド数分だけ横ブロック分割し、行列 *B* は行列 1 個を単位として使用します。よって、各行列のサイズを $n \times n$ 、プロセッサ数を $pnum$ 、行列の各成分を `double` 型とすれば、行列 *A* と行列 *C* のページサイズは $n \times n / pnum \times \text{sizeof}(\text{double})$ 、行列 *B* のページサイズは $n \times n \times \text{sizeof}(\text{double})$ に設定するのが理想です^{*1}。具体的には、以下のように確保することができます：

```
int64_t a_addr, b_addr, c_addr;

DMI_mmap(&a_addr, n / pnum * n * sizeof(double), pnum, NULL);
DMI_mmap(&b_addr, n * n * sizeof(double), 1, NULL);
DMI_mmap(&c_addr, n / pnum * n * sizeof(double), pnum, NULL);
```

■ 2-3 補足

2-3-1 ページの実体が確保されるタイミング

`DMI_mmap(...)` は、呼び出された時点ですぐに、この `DMI_mmap(...)` で呼び出した DMI プロセスのメモリプールにページの実体を確保するわけではありません。`DMI_mmap(...)` を呼び出した時点で生成されるのは、キャッシュ管理のために DMI が使用するページテーブルだけであって、実際にページの実体がメモリプールに確保されるのは、そのページに対して初めて `DMI_read(...)`/`DMI_write(...)` が行われた時点です。したがって、DMI プロセス *i* が提供しているメモリプールの容量を大幅に上回るようなサイズのグローバルメモリを、その DMI プロセス *i* が `DMI_mmap(...)` で確保することも十分に可能です。Fig.??に示すように、`DMI_mmap(...)` によってひとまず確保したあとで、残りの DMI プロセ

^{*1} 後述する選択的キャッシュ read/write の仕組みを利用方法によっては、行列 *C* のページサイズは $n \times n \times \text{sizeof}(\text{double})$ とする方がよいこともあります。

スたちが自分の担当領域に関して `DMI_read(...)/DMI_write(...)` するようにプログラムしておけば、グローバルメモリを構成するメモリの実体が各 DMI プロセス上のメモリプールに分散配置することができます。この仕組みを利用すると、多数の DMI プロセスを参加させて巨大なメモリプールを作り出し、1DMI プロセスの物理メモリ量をはるかに超えるような巨大なグローバルメモリを実現することもできます。これは遠隔スワップと呼ばれている技術で、近年、ディスクアクセスよりも他のノードのメモリへのアクセスの方が高速化していることをふまえ、ディスクスワップの代替手段として着目されている技術です。ただし、DMI は、遠隔スワップシステムとして作り込んでいるわけではないので、それを本業としているような処理系と比較すると性能は良くないと思われます。

2-3-2 `DMI_mmap(...)/DMI_munmap(...)` は重い

`DMI_mmap(...)/DMI_munmap(...)` が呼ばれたとき、DMI は暗黙的に全 DMI プロセスで同期して、必要なデータ管理を行います。つまり、`DMI_mmap(...)/DMI_munmap(...)` は、暗黙的に全 DMI プロセスとの同期を伴うような重い操作です。よって、頻繁な `DMI_mmap(...)/DMI_munmap(...)` の呼び出しは避けるべきであり、可能な限り、`DMI_mmap(...)` で確保したグローバルメモリをアプリケーション内で再利用することが望まれます。同様のことは、通常の共有メモリ環境の `mmap(...)/munmap(...)` にもいえますが、共有メモリ環境では、`mmap(...)/munmap(...)` の頻繁な呼び出しを避けるための賢いメモリ確保/解放の API として `malloc(...)/free(...)` が用意されているわけです*2。

2-3-3 `DMI_mmap(...)/DMI_munmap(...)` を行うタイミング

2-3-4 構造体の特定のメンバへの read/write

3 グローバルメモリの read/write

■ 3-1 API

3-1-1 API

グローバルメモリに対しては `DMI_read(...)/DMI_write(...)` で read/write を行います。正確な API は以下のとおりです：

- `int32_t DMI_read(int64_t addr, int64_t size, void *buf, int8_t mode, DMI_local_status_t *status);`

グローバルメモリアドレス `addr` から `size` バイトをローカルメモリアドレス `buf` に read します。`mode` には、`DMI_INVALIDATE_READ`、`DMI_UPDATE_READ`、`DMI_GET_READ` のいずれか

*2 共有メモリ環境での `malloc(...)` は、要求されるサイズに応じて `brk(...)` もしくは `mmap(...)` を呼び出すように実装されていますが、`malloc(...)` のたびにこれらのシステムコールを呼んでいるわけではなく、いったん確保した領域を上手に再利用するように実装されています。DMI にも、共有メモリ環境での `malloc(...)/free(...)` に相当するような、賢いメモリ確保/解放を行うための `DMI_malloc(...)/DMI_free(...)` が存在することが望まれますが、現状の DMI にはまだ実装してありません。

のモードを指定でき、この `DMI_read(...)` に伴ってキャッシュをどう管理するかを明示的に指定することができます。addr がページ境界にアラインされている必要はありません。また、`[addr,addr+size)` のアドレス領域が複数のページにまたがることもできます。status を利用することで非同期呼び出しにできます。

- `int32_t DMI_write(int64_t addr, int64_t size, void *buf, int8_t mode, DMI_local_status_t *status);`

ローカルメモリアドレス `buf` から `size` バイトをグローバルメモリアドレス `addr` に write します。mode には、`DMI_EXCLUSIVE_WRITE`、`DMI_PUT_WRITE` のいずれかのモードを指定でき、この `DMI_write(...)` に伴ってキャッシュをどう管理するかを明示的に指定することができます。addr がページ境界にアラインされている必要はありません。また、`[addr,addr+size)` のアドレス領域が複数のページにまたがることもできます。status を利用することで非同期呼び出しにできます。

mode の値に応じてキャッシュ管理を明示的に指定できる機能は、選択的キャッシュ read/write と呼ばれ、DMI を特徴づける非常に重要な機能です。選択的キャッシュ read/write に関しては、??節で解説します。

3-1-2 高性能なプログラムを記述するうえでのポイント

??で述べたように、DMI で高性能なプログラムを記述するためには、`DMI_read(...)/DMI_write(...)` を呼び出す回数を最小限に抑制すべきです。たとえば、行列行列積のプログラムにおいて、各プロセスが部分行列行列積を求めるときに、

```
double a, b, c;

for(i = 0; i < n / pnum; i++)
{
    for(k = 0; k < n; k++)
    {
        for(j = 0; j < n; j++)
        {
            DMI_read(a_addr + targ.rank * n / pnum * n * sizeof(double) + (i * n + k) * sizeof(double),
            DMI_read(b_addr + (k * n + j) * sizeof(double), sizeof(double), &b, DMI_INVALIDATE_RE
            c = a * b;
            DMI_write(c_addr + targ.rank * n / pnum * n * sizeof(double) + (i * n + j) * sizeof(d
        }
    }
}
```

のように記述するのはオーバーヘッドが非常に大きくなります。もちろん、行列 *A* と行列 *C* のページサイズを `n*n/pnum*sizeof(double)` に、行列 *B* のページサイズを `n*n*sizeof(double)` に指定しておけば、各行列に関して、1 回目の `DMI_read(...)/DMI_write(...)` だけがページフォルトを引き起こして、その DMI プロセスのメモリプールにページがキャッシュされるので、以降の `DMI_read(...)/DMI_write(...)` はローカルに完了します。しかし、`DMI_read(...)/DMI_write(...)` の関数呼び出しのオーバーヘッドが無視できない

上, DMI では, DMI プロセスの参加/脱退や各種の高度なチューニングを実現するために, `DMI_read(...)/DMI_write(...)` の内部で, さまざまな検査を行っています。また, ローカルメモリとグローバルメモリの間のコピーがその都度発生するのも重いと思われます。したがって, 上述のようなコードではなく,

```
double *local_a, *local_b, *local_c;

local_a = (double*)malloc(n / pnum * n * sizeof(double));
local_b = (double*)malloc(n * n * sizeof(double));
local_c = (double*)malloc(n / pnum * n * sizeof(double));

DMI_read(a_addr + targ.rank * n / pnum * n * sizeof(double), n / pnum * n * sizeof(double), local_a);
DMI_read(b_addr, n * n * sizeof(double), local_b, DMI_INVALIDATE_READ, NULL);
for(i = 0; i < n / pnum; i++)
{
    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];
        }
    }
}
DMI_write(c_addr + targ.rank * n / pnum * n * sizeof(double), n / pnum * n * sizeof(double), local_c);
```

のように記述するのが理想です^{*3}。要するに, 高性能な分散プログラミング処理系には一般的にいえることですが, DMI では, ローカルとグローバルを明確に区別したプログラミングを行うことが重要です。

■ 3-2 コンシステンシモデル

3-2-1 直観的な説明

`read/write` ベースの並列処理系においては, コンシステンシモデルが定義される必要があります。本節では DMI のコンシステンシモデルについて説明しますが, やや専門的な知識が必要ですので, よくわからない場合には以下のような挙動をするものだと考えてください:

- `[addr, addr+size)` のアドレス領域が 1 個のページに収まるような同期的な `DMI_read(...)/DMI_write(...)` で `read/write` されるデータは, プログラマが(おそらく)意図したであろうデータと一致します。よって, 特に難しいことを考える必要は(おそらく)ありません。
- 非同期 `DMI_read(...)/DMI_write(...)` は, 呼び出したその時点で関数が返るため, いつのデータが `read/write` されるかはわかりません。意図したデータを `read/write` するには, 別途同期などを行う必要があります。

^{*3} ちなみに, 3 重の for ループを `ikj` の順に行っているのは, CPU のキャッシュヒット率を高めるためです。一般に, `ijk` ループで記述するよりもグッと速くなるはずですが, よりローカルな計算を高速化するためにはキャッシュブロッキングなどの工夫が考えられます。

- `[addr,addr+size)` のアドレス領域が複数のページにまたがるような同期的な `DMI_read(...)/DMI_write(...)` に関しては、その `DMI_read(...)/DMI_write(...)` がページ単位での小 `DMI_read(...)/小 DMI_write(...)` に分割され、それら小 `DMI_read(...)/小 DMI_write(...)` が任意の順序で独立に発行された結果になります。

3-2-2 正確な説明

正確には、DMI では以下のようなコンシステンシモデルを採用しています：

- `[addr,addr+size)` のアドレス領域が 1 個のページに収まるような同期的な `DMI_read(...)/DMI_write(...)` に関しては、Sequential Consistency が保証されます。すなわち、複数の DMI プロセスが `DMI_read(...)/DMI_write(...)` を発行したとき、これらすべての `DMI_read(...)/DMI_write(...)` に関するある逐次的な順序 O が存在して、全 DMI プロセスが、これらすべての `DMI_read(...)/DMI_write(...)` が順序 O で行われたかのように結果を観測することが保証されます。ただし、同一 DMI プロセス内での `DMI_read(...)/DMI_write(...)` の順序は、そのまま順序 O に反映されているものとします。言い換えると、異なる DMI プロセスで発行される `DMI_read(...)/DMI_write(...)` に関しては、その順序に関してインタリーブが許されますが、そのインタリーブのされ方は全 DMI プロセスによって同じように観測されることが保証されます。
- `[addr,addr+size)` のアドレス領域が複数のページにまたがるような同期的な `DMI_read(...)/DMI_write(...)` に関しては、その `DMI_read(...)/DMI_write(...)` がページ単位での小 `DMI_read(...)/小 DMI_write(...)` に分割され、それら小 `DMI_read(...)/小 DMI_write(...)` が任意の順序で独立に発行されたものと見なされます。
- 非同期 `DMI_read(...)/DMI_write(...)` に関しては、特にコンシステンシは保証されません。

Sequential Consistency は分散処理系が保証することが可能なコンシステンシモデルのなかで最も強いコンシステンシモデルであり、プログラマ視点では極めて直観に従う `DMI_read(...)/DMI_write(...)` を実現します。一般的には、コンシステンシモデルが強ければ強いほどプログラミングは直観的になりますが、保証すべきコンシステンシの制約が強くなるため、必要な通信量が増大して性能は鈍ります。そのため、最近の分散共有メモリでは、性能上の要請から、より緩和されたコンシステンシモデルが採用される場合が多いです。このような状況のなかで、DMI が敢えて Sequential Consistency という強いコンシステンシモデルを採用しているのは、非同期な `DMI_read(...)/DMI_write(...)` という手段を提供することで、プログラマが明示的にコンシステンシモデルを緩和できると考えたためです。言い換えると、DMI としては強いコンシステンシモデルを採用しつつも、アプリケーションの要請に応じてコンシステンシモデルを明示的に緩和するための手段を提供しているということです。

3-2-3 複数のページにまたがる `DMI_read(...)/DMI_write(...)`

複数のページにまたがる同期的な `DMI_read(...)/DMI_write(...)` に関して補足しておきます。上記の説明を言い換えると、複数のページにまたがる同期的な `DMI_read(...)/DMI_write(...)` は、

それが対象とする各ページに対して独立かつ非同期に小 DMI_read(...)/小 DMI_write(...) を発行したあと、それらすべての小 DMI_read(...)/小 DMI_write(...) の完了を DMI_wait(...) で待ち、すべての小 DMI_read(...)/小 DMI_write(...) が完了した時点で返るという仕様になっています。したがって、たとえば、先頭アドレスが `addr` のグローバルメモリのページサイズが `sizeof(int)` のときに、DMI プロセス 0 が、

```
int buf[] = {0, 0, 0, 0};
DMI_write(addr, 4 * sizeof(int), buf, DMI_EXCLUSIVE_WRITE, NULL);
プロセス 1 と同期;
DMI_read(addr, 4 * sizeof(int), buf, DMI_GET_READ, NULL);
for(i = 0; i < 4; i++)
{
    printf("%d ", buf[i]);
}
```

というコードを実行し、DMI プロセス 1 が、

```
int buf[] = {1, 1, 1, 1};
DMI_write(addr, 4 * sizeof(int), buf, DMI_EXCLUSIVE_WRITE, NULL);
プロセス 0 と同期;
DMI_read(addr, 4 * sizeof(int), buf, DMI_GET_READ, NULL);
for(i = 0; i < 4; i++)
{
    printf("%d ", buf[i]);
}
```

というコードを実行したとき、DMI プロセス 0 が 1 1 1 0 と出力し、DMI プロセス 1 が 0 1 0 0 と出力するような可能性があるということです。複数のページにまたがって DMI_read(...)/DMI_write(...) を呼び出す際には、必要に応じて明示的に別途同期を行ってください。

なお、行列行列積の例の場合、最後に DMI プロセス 0 が行列 *C* を DMI_read(...) する部分は、複数のページにまたがった DMI_read(...) が行われています。この場合、直前のバリア操作によって、DMI_read(...) されるデータが必ず意図したものになることが保証されています。

■ 3-3 選択的キャッシュ read/write

3-3-1 キャッシュ管理の仕組み

DMI では、各 DMI_read(...)/DMI_write(...) のたびに、ページ単位での何らかのキャッシュ管理が行われますが、どのようなキャッシュ管理を行うかを各 DMI_read(...)/DMI_write(...) の引数 `mode` として明示的に指定することができます。引数 `mode` に何を指定できるかを説明する前に、DMI におけるキャッシュ管理の概要について説明します。キャッシュ管理は各ページに関して独立に行われるため、以降では、あるページ *p* に着眼して話を進めます。

DMI プロセス *i* のメモリプールにページ *p* の実体が保持されているとき、DMI プロセス *i* はページ *p* をキャッシュしているといえます。キャッシュの状態としては、INVALIDATE 型キャッシュと

UPDATE 型キャッシュの 2 種類が存在します。INVALIDATE 型キャッシュは、ページ p が更新されたときに無効化されるキャッシュです。UPDATE 型キャッシュは、ページ p が更新されたときに更新部分のデータが送り付けられることで、つねに最新状態に保たれるキャッシュです。また、各ページ p にはオーナーと呼ばれる DMI プロセスがつねに 1 個だけ存在します。オーナーは、ページ p に関するキャッシュ管理の権限を握っており、つねに最新状態のページ p をメモリプールに保持していて、他の DMI プロセスがページ p をどのような状態でキャッシュしているのかをすべて把握しています。

DMI プロセス i 上の DMI スレッドによる `DMI_read(...)` がページフォルトを引き起こすのは、DMI プロセス i がページ p をキャッシュしていない場合です。DMI プロセス i 上の DMI スレッドによる `DMI_write(...)` がページフォルトを引き起こすのは、DMI プロセス i がオーナーでない場合、またはページ p をキャッシュしている DMI プロセスが DMI プロセス i 以外にも存在する場合です。そして、DMI プロセス i でページフォルトが発生した場合には、DMI プロセス i はオーナーに対してページフォルトを通知し、オーナーが適切なキャッシュ管理を行うことでページフォルトが解決されます。このときに、INVALIDATE 型キャッシュでキャッシュしている DMI プロセスに対する無効化要求や、UPDATE 型キャッシュでキャッシュしている DMI プロセスに対する更新データの送り付けなどが、オーナーによって行われます。オーナーはどれかの DMI プロセスに固定されているわけではなく、そのページに対して read/write が行われるにつれて、必要に応じて動的に変化します。また、`DMI_mmap(...)` でグローバルメモリを確保した直後では、そのグローバルメモリ内のすべてのページのオーナーはその `DMI_mmap(...)` を発行したノードに設定されます。先ほど、オーナーはつねにページの最新状態を保持していると書きましたが、実は `DMI_mmap(...)` した直後だけは例外で、オーナーはページの実体を保持していません。??で述べたように、ページの実体が確保されるのは、各ページに対して初めての `DMI_read(...)/DMI_write(...)` が発行された時点です。

今の説明からわかるように、DMI では、同一 DMI プロセス上の複数の DMI スレッドが 1 個のメモリプールを共有する構造になっています。よって、DMI プロセス i 上の DMI スレッド t_1 と DMI スレッド t_2 があったとき、あるページ p が、DMI スレッド t_1 の read/write によって DMI プロセス i のメモリプールにキャッシュされていれば、そのあとで DMI スレッド t_2 がページ p に対して read/write したときにページフォルトは発生しません。このように、DMI では、マルチコアレベルの並列性を活用するようなハイブリッドプログラミングを透過的に実現しています。

以上のようなキャッシュ管理においては、INVALIDATE 型キャッシュでキャッシュするのか UPDATE 型キャッシュでキャッシュするのか、オーナーを動的に変化させるのかどうかなどに自由度があります。これを明示的に指定できる機能が選択的キャッシュ read/write です。

3-3-2 選択的キャッシュ read/write の詳細

`DMI_read(...)` の引数 `mode` には、`DMI_INVALIDATE_READ`、`DMI_UPDATE_READ`、`DMI_GET_READ` のいずれかのモードを指定でき、ページ p の read に伴ってそれぞれ以下のようなキャッシュ管理を行います：

DMI_INVALIDATE_READ DMI プロセス i はオーナーに対してページ p を要求し、ページ p を INVALIDATE 型キャッシュとして DMI プロセス i にキャッシュします (Fig.??(A))。

DMI_UPDATE_READ DMI プロセス i はオーナーに対してページ p を要求し、ページ p を

UPDATE 型キャッシュとして DMI プロセス i にキャッシュします (Fig.??(B)).

DMI_GET_READ DMI プロセス i はオーナーに対してページ p の中で read したい部分のデータのみを要求して、そのデータのみオーナーから送信してもらいます。ページ p はキャッシュしません。すなわち、get 操作を行います (Fig.??(C)).

DMI_write(...) の引数 mode には、DMI_EXCLUSIVE_WRITE, DMI_PUT_WRITE のいずれかのモードを指定でき、ページ p の write に伴ってそれぞれ以下のようなキャッシュ管理を行います：

DMI_EXCLUSIVE_WRITE DMI プロセス i は、まずオーナーからページ p のオーナー権を奪ってきて、DMI プロセス i が新しいオーナーになります。次に、INVALIDATE 型キャッシュでページ p をキャッシュしている DMI プロセスに対して無効化要求を送信してキャッシュを無効化し、UPDATE 型キャッシュでページ p をキャッシュしている DMI プロセスに対して更新部分のデータを送り付けることでキャッシュを最新に保ちます (Fig.??(D)).

DMI_PUT_WRITE DMI プロセス i は、write したい部分のデータをオーナーに対して送信します。すると、オーナーはそのデータに基づいてページ p を更新したあと、INVALIDATE 型キャッシュでページ p をキャッシュしている DMI プロセスに対して無効化要求を送信してキャッシュを無効化し、UPDATE 型キャッシュでページ p をキャッシュしている DMI プロセスに対して更新部分のデータを送り付けることでキャッシュを最新に保ちます。このときオーナーは変化しません。つまり put 操作を行います (Fig.??(E)).

3-3-3 mode を選択する際の一般論

mode としてどの値を指定するのが好ましいかに関して、具体例を示す前にまず一般論を説明します。

まず DMI_read(...) についてですが、ページ p を今の 1 回だけ read できれば十分であって、今後ページ p が更新されるまでの間に read する可能性が低い場合には、DMI_GET_READ モードが適しています。また、DMI_GET_READ モードは get 操作であり、ページサイズにかかわらずページの一部のみを read することができるため、read する範囲がページのごく一部である場合には DMI_GET_READ モードが適しています。今後ページ p が更新されるまでの間に read する可能性があり、かつページ p に対する操作として read よりも write の比率が高い場合には、DMI_INVALIDATE_READ モードが適しています。今後ページ p が更新されるまでの間に read する可能性があり、かつページ p に対する操作として write よりも read の比率が高い場合には、DMI_UPDATE_READ モードが適しています。このように DMI では、INVALIDATE 型のキャッシュプロトコルと UPDATE 型のキャッシュプロトコルを各 DMI_read(...) の粒度でハイブリッドさせることができます。当然、キャッシュしている DMI プロセスが多いほど、特に DMI_UPDATE_READ モードでキャッシュしている DMI プロセスが多いほど、DMI_write(...) に伴うキャッシュ管理が重たくなります。DMI_GET_READ モードで DMI_read(...) しておけば、DMI_write(...) のときには無効化要求や更新要求が必要ありません。このような事情を勘案し、アプリケーションにとって最も効率的な mode を選択する必要があります。

次に DMI_write(...) についてですが、write した DMI プロセスにオーナーを割り当てたい場合には DMI_EXCLUSIVE_WRITE モードが適しています。データ並列なアプリケーションなどにおいて、意図するデータ分散（オーナー割り当て）を実現するためには、DMI_EXCLUSIVE_WRITE モードを使

うことで、「このページの実体はこの DMI プロセスのメモリプールに格納したい」ということを明示的に指示できます。また、計算規模が動的に変化するような動的なアプリケーションにおいては、静的にデータ分散を決め打つだけでは次第にデータのローカリティが悪くなることがあり、実行開始時に計算規模が予測できない場合にはそもそも静的に最適なデータ分散を指示すること自体が困難です。このような場合には、DMI_EXCLUSIVE_WRITE モードを使うことで、実際に write した DMI プロセスにオーナーを移動させて割り当てることができ、動的にデータのローカリティを改善できます。つまり、実際のアクセスパターンに追隨して動的にデータのローカリティを適応させることができます。しかしながら、DMI_EXCLUSIVE_WRITE モードによってオーナーを移動させることには欠点もあります。前述のように、DMI におけるキャッシュ管理では、ページフォルトが発生したときオーナーにページフォルトを通知しますが、このとき内部的にはオーナーの「追跡」が行われます。オーナーがどう移動しようともこのオーナーの追跡は必ず成功しますが、オーナーの移動回数が多いほど追跡に要するメッセージ数が増えます。また、オーナーはつねにページをキャッシュしている必要があるため、新しいオーナーになる DMI プロセスがページをキャッシュしていない場合には、古いオーナーが新しいオーナーに対して最新ページを送信することも必要になります。このように、オーナーの移動はある程度重い処理を伴うため、DMI_EXCLUSIVE_WRITE モードによって頻繁にオーナーを移動させるのは避けるべきです。ただし、これは DMI_EXCLUSIVE_WRITE モードを頻繁に呼び出すことが悪いという意味ではありません。同一ノードが連続して DMI_EXCLUSIVE_WRITE モードを呼び出すのであれば、2 回目以降の DMI_EXCLUSIVE_WRITE モードはオーナーの移動を引き起こしません。むしろ、動的なアプリケーションにおいてデータのローカリティを改善するために、「データをつねに DMI_EXCLUSIVE_WRITE モードで write しておく」ということは良く行われます。一方で、すでに何らかのデータ分散（オーナー割り当て）が決定しており、そのオーナー割り当てを崩すことなく write したい場合や、DMI_EXCLUSIVE_WRITE モードで write するとオーナーの頻繁な移動を引き起こす可能性がある場合には、DMI_PUT_WRITE モードが適しています。また、DMI_PUT_WRITE モードは put 操作であり、ページサイズにかかわらずページの一部だけを write することができるため、write する範囲がページのごく一部であるような場合には DMI_PUT_WRITE モードが適しています。しかし、DMI_PUT_WRITE モードは、write する際に更新部分のデータをオーナーに送信する必要があるため、更新部分のデータが大きかつそのノードがページをキャッシュしているならば、DMI_EXCLUSIVE_WRITE モードの方が通信量が少なく速い場合があります。いずれにせよ、各アプリケーションごとにデータのローカリティ、オーナーの移動の頻度、write が引き起こすであろうデータ通信量などを分析したうえで、最適な mode を選択することが重要です。

このように、この選択的キャッシュ read/write は、read/write に伴って引き起こされる通信を支配するため、DMI プログラムを高性能化する際の 90% は、適切な選択的キャッシュ read/write を選択できるかどうかにかかっているといっても過言ではありません。

3-3-4 具体例（遠隔スワップ）

??節で述べたように、遠隔スワップを実現するために巨大なグローバルメモリを DMI_mmap(...) する場合を考えてみます。このとき、1 個の DMI プロセスのメモリプールには収まりきらないサイズのグローバルメモリを確保するわけなので、1 個の DMI プロセスがグローバルメモリ全体を DMI_write(...) で初期化するのは非効率的です。DMI は、メモリプールの使用量が、DMI プロセス起動時に指定された

サイズを超えた場合には、適宜ページを置換することでメモリプールの使用量を下げようとするため、このようなことをしても一応正しく動作はしますが、ページ置換が頻発するため明らかに性能は落ちます。このような場合には、巨大なグローバルメモリのどの部分をどの DMI プロセスのメモリプールに格納するか（つまり各ページのオーナーをどの DMI プロセスにするか）を決めておき、DMI_mmap(...) したあとで、複数の DMI プロセスが、自分の担当するページに関して DMI_EXCLUSIVE_WRITE モードで DMI_write(...) を発行します。具体的には、n バイトのグローバルメモリを pnum 個の DMI プロセスのメモリプールに分散配置する場合、以下のようにグローバルメモリの初期化を行います：

```
typedef struct targ_t
{
    int32_t rank; /* DMI スレッドのランク */
    int32_t pnum; /* DMI スレッドの個数 */
    int32_t page_size; /* ページサイズ */
    int64_t mem_addr; /* グローバルメモリの先頭アドレス */
}targ_t;

void DMI_main(int argc, char **argv)
{
    DMI_node_t node;
    DMI_node_t *nodes;
    DMI_thread_t *threads;
    targ_t targ;
    int32_t i, j, node_num, thread_num, rank, pnum;
    int64_t targ_addr, mem_addr, page_size, size;

    if(argc != 4)
    {
        fprintf(stderr, "usage : %s node_num thread_num size\n", argv[0]);
        exit(1);
    }

    node_num = atoi(argv[1]); /* 使用する DMI プロセスの個数 */
    thread_num = atoi(argv[2]); /* 各 DMI プロセスに生成する DMI スレッドの個数 */
    size = atoll(argv[3]); /* 確保するグローバルメモリのサイズ */
    pnum = node_num * thread_num; /* 生成する DMI スレッドの個数 */
    page_size = size / pnum; /* ページサイズ */
    nodes = (DMI_node_t*)malloc(node_num * sizeof(DMI_node_t));
    threads = (DMI_thread_t*)malloc(pnum * sizeof(DMI_thread_t));
    for(i = 0; i < node_num; i++) /* node_num 個の DMI プロセスの参加を許可 */
    {
        DMI_poll(&node);
        if(node.state == DMI_OPEN)
        {
            DMI_welcome(node.dmi_id);
            nodes[i] = node;
        }
    }

    DMI_mmap(&targ_addr, sizeof(targ_t), pnum, NULL);
    DMI_mmap(&mem_addr, page_size, pnum, NULL); /* ページサイズ page_size のページを pnum 個持つグ
ローバルメモリ B を確保。この時点ではページの实体はまだ割り当てられていない */
    for(rank = 0; rank < pnum; rank++) /* 各 DMI スレッドに渡す引数を仕込む */
    {
        targ.rank = rank;
```

```

    targ.pnum = pnum;
    targ.mem_addr = mem_addr;
    targ.page_size = page_size;
    DMI_write(targ_addr + rank * sizeof(targ_t), sizeof(targ_t), &targ, DMI_EXCLUSIVE_WRITE, NULL);
}
rank = 0;
for(i = 0; i < node_num; i++) /* 各 DMI プロセスに thread_num 個の DMI スレッドを生成 */
{
    node = nodes[i];
    for(j = 0; j < thread_num; j++)
    {
        DMI_create(&threads[rank], node.dmi_id, targ_addr + rank * sizeof(targ_t), NULL);
        rank++;
    }
}
...;
}

int64_t DMI_thread(int64_t targ_addr)
{
    targ_t targ;
    int i;
    char *buf;

    DMI_read(targ_addr, sizeof(targ_t), &targ, DMI_GET_READ, NULL); /* この DMI スレッドに対する
    引数を読む */
    buf = (char*)malloc(targ.page_size);
    for(i = 0; i < targ.page_size; i++) /* 0 で埋めたローカルメモリを作る */
    {
        buf[i] = 0;
    }
    DMI_write(targ.mem_addr + targ.rank * targ.page_size, targ.page_size, buf, DMI_EXCLUSIVE_WRITE, NULL);
    /* この DMI スレッドが担当する分のグローバルメモリを EXCLUSIVE モードで DMI_write(...) する . このとき初めてページの
    実体が割り当てられる */
    ...;
}

```

これにより、意図したとおりに、複数の DMI プロセスに対してページの実体を分散させることができます。そして、以降でこのグローバルメモリに write する際につねに DMI_PUT_WRITE モードを使うようにすれば、これらのデータ分散を崩すことなく処理を進められます。

3-3-5 具体例（行列行列積）

行列行列積のプログラムの最後の部分で、行列 C をプロセス 0 に gather する処理を考えてみます。このとき主に考え方は 2 とおりあります。第 1 の方法は、各 DMI スレッドが部分行列 C_i を DMI_EXCLUSIVE_WRITE モードで DMI_write(...) し、全 DMI スレッドで同期をとったあと、最後に DMI スレッド 0 が行列 C 全体を DMI_GET_READ モードで DMI_read(...) する方法です。つまり、プログラムとしては、

```

int64_t DMI_thread(int64_t targ_addr)
{
    ...;
}

```

```

for(i = 0; i < nn; i++) /* 部分行列行列積 Ci=Ai*B を計算 */
{
    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];
        }
    }
}
DMI_write(c_addr + targ.rank * nn * n * sizeof(double), nn * n * sizeof(double), local_c, DMI_E
分行列 Ci をグローバルメモリに書き込む */
全 DMI プロセスでバリア操作;
if(targ.rank == 0)
{
    DMI_read(c_addr, n * n * sizeof(double), original_c, DMI_GET_READ, NULL); /* 行列 C
全体を読み込む */
    ...;
}
...;
}

```

のようになります。なお、この `DMI_read(...)` では、`DMI_GET_READ`、`DMI_INVALIDATE_READ`、`DMI_UPDATE_READ` のいずれのモードを使っても、各 DMI スレッドが存在する DMI プロセスのメモリプールから DMI スレッド 0 が存在する DMI プロセスのメモリプールに行列 C 全体が転送されるため、性能に違いはありません。この場合、行列 C のデータの通信は DMI スレッド 0 が `DMI_read(...)` を発行した時点で起こります。

第 2 の方法は、各 DMI スレッドが部分行列 C_i を `DMI_PUT_WRITE` モードで `DMI_write(...)` し、全 DMI スレッドで同期をとったあと、最後に DMI スレッド 0 が行列 C 全体を `DMI_GET_READ` モードで `DMI_read(...)` する方法です。つまり、プログラムとしては、

```

int64_t DMI_thread(int64_t targ_addr)
{
    ...;
    for(i = 0; i < nn; i++) /* 部分行列行列積 Ci=Ai*B を計算 */
    {
        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];
            }
        }
    }
    DMI_write(c_addr + targ.rank * nn * n * sizeof(double), nn * n * sizeof(double), local_c, DMI_P
分行列 Ci をグローバルメモリに書き込む */
    全 DMI プロセスでバリア操作;
    if(targ.rank == 0)
    {
        DMI_read(c_addr, n * n * sizeof(double), original_c, DMI_GET_READ, NULL); /* 行列 C
全体を読み込む */
    }
}

```

```

        ...;
    }
    ...;
}

```

のようになります。この場合には、行列 C のデータの通信は各 DMI スレッドが `DMI_write(...)` を発行した時点で起こり、DMI スレッド 0 による `DMI_read(...)` はローカルに完了します。

さて、この処理においてはどちらの方法をとってもほとんど性能上の差異はありませんが、厳密に言えば、以下のような理由で第 2 の方法の方が優れています。第 1 の方法の場合には、各 DMI スレッドが `DMI_EXCLUSIVE_WRITE` モードで `DMI_write(...)` するため、行列 C のページサイズは `n*n/pnum*sizeof(double)` に設定する必要があります。一方で、第 2 の方法の場合には、各 DMI スレッドが `DMI_PUT_WRITE` モードで `DMI_write(...)` するため、この `DMI_write(...)` においてはページサイズが関係ありません。よって、行列 C のページサイズを `n*n*sizeof(double)` にとれるため、キャッシュ管理のオーバーヘッドが少しだけ小さくて済みます。

■ 3-4 より高度なチューニング

3-4-1 必要なページをキャッシュから落とさない

各 DMI プロセスは、必要に応じてページをメモリプール内にキャッシュしますが、当然、際限なくメモリプールにページをキャッシュし続けられるわけではありません。使用可能なメモリプールの量は、各 DMI プロセスの起動時に `-s` オプションによって指定されます（指定しない場合のデフォルト値は 4GB です）。よって、DMI では、メモリプールの使用量が飽和した場合に自動的にページの追い出しを行い、メモリプールの使用量を下げます。正確には、DMI プロセス i は、追い出し総量が一定量に達するまで、以下のページ置換アルゴリズムに従ってページの追い出しを行います。

- ページサイズの大きい順にページを追い出します。
- 同じページサイズのページに関しては、実装上の負荷の大小を根拠として、以下の優先度順でページを追い出します。
 - (1) DMI プロセス i がオーナーではないページ
 - (2) DMI プロセス i がオーナーであり、DMI プロセス i 以外にそのページをキャッシュしている DMI プロセスが存在するページ
 - (3) DMI プロセス i がオーナーであり、DMI プロセス i 以外にそのページをキャッシュしている DMI プロセスが存在しないページ
- ページの追い出し先としては、追い出しの負荷が小さくなるであろう DMI プロセスを選択します。

このように DMI では独自のページ置換アルゴリズムに従ってページの追い出しを行います。以下の API を利用することで、プログラマが明示的に特定のページを追い出し禁止し、必要なページが DMI によって勝手にキャッシュから外されるのを防ぐことができます：

- `int32_t DMI_save(int64_t addr, int64_t size) :`
 グローバルメモリアドレス `addr` から `size` バイトの範囲に含まれるページを追い出し禁止にします。

- `int32_t DMI_unsave(int64_t addr, int64_t size) :`

グローバルメモリアドレス `addr` から `size` バイトの範囲に含まれるページを追い出し可能にします。

初期的にはすべてのページが追い出し可能に設定されています。よって、`DMI_unsave(...)` は、`DMI_save(...)` によっていったん追い出し禁止に変更したページを、追い出し可能に戻すために使います。

3-4-2 データ転送の負荷を分散する

分散プログラミング処理系においては、Broadcast などの集合通信を効率化することが重要です。DMI では、集合通信のための特別な API は提供していませんが、特定の DMI プロセスに対するデータ転送の要求の集中を自動的に検知して、動的にデータ転送の負荷を分散する機能を実装しています。ただし、この機能をうまく利用するためには、プログラム上の注意が必要なので説明します。

DMI においてデータ転送を動的に負荷分散する機能は、Broadcast や Allgather など特定の決まったパターンの通信だけを最適化しようとするものではなく、もっと一般に、特定の DMI プロセスに対するデータ転送の要求が集中しているときにそれらのデータ転送を負荷分散する機能です。ただし、ここでは説明を簡単化するため、大きなページに関して Broadcast が起きる場合を例にして説明します。DMI で、ページ p に関して Broadcast に相当する通信が起きるのは、1 個の DMI スレッド t がページ p に（たとえば）`DMI_EXCLUSIVE_WRITE` モードで `DMI_write(...)` したあと、全 DMI スレッドがページ p を（たとえば）`DMI_INVALIDATE_WRITE` モードで `DMI_read(...)` する場合です。このとき、前述のキャッシュ管理の説明に従うと、DMI スレッド t が所属する DMI プロセス i 以外の DMI プロセスに所属する全 DMI スレッドの `DMI_read(...)` はすべてページフォルトを引き起こし、これらのページフォルトはオーナーである DMI プロセス i へと通知されます。そして、オーナーである DMI プロセス i は、これらすべてのページフォルトを逐次的に処理して、Fig.?? に示すように、ページフォルトを引き起こした各 DMI プロセスに逐次的にページを転送します。しかし、このようにオーナーがすべての DMI プロセスに対して逐次的にページを転送するのは、データ転送の負荷がオーナーに集中しており非効率的です。

そこで、DMI では、データ転送の負荷がオーナーに集中していることを検知した場合には、ページフォルトを起こした各 DMI プロセスからのページ p の転送要求の一部を、すでにページ p をキャッシュしているノードに委任することで、Fig.?? に示すように、全体としてページ転送を木構造化させます。詳細なアルゴリズムの説明は省きますが、ページサイズが大きいページに関する Broadcast に相当する通信が起きた場合、ページ転送は、Fig.?? に示すような Binomial Tree 状に効率的に負荷分散されます。また、Broadcast のような定型的な集合通信でなくとも、データ転送の負荷が検知されれば動的に負荷分散が行われます。

以上の説明からわかるように、このデータ転送の動的負荷分散の機能を利用するためには、オーナー以外の DMI プロセスがページをキャッシュしていることが条件になります。したがって、たとえば、行列行列積のプログラムで行列 B を Broadcast する部分を、

```
if(targ.rank == 0)
{
    ...;
    DMI_write(b_addr, n * n * sizeof(double), original_b, DMI_EXCLUSIVE_WRITE, NULL);
```

```

    }
    全 DMI プロセスでバリア操作;
    ...;
    local_b = (double*)malloc(n * n * sizeof(double));
    DMI_read(b_addr, n * n * sizeof(double), local_b, DMI_GET_READ, NULL);

```

のように、DMI_GET_READ モードの DMI_read(...) を使って記述してしまうと、データ転送の動的負荷分散が行われません。そこで、

```

if(targ.rank == 0)
{
    ...;
    DMI_write(b_addr, n * n * sizeof(double), original_b, DMI_EXCLUSIVE_WRITE, NULL);
}
    全 DMI プロセスでバリア操作;
    ...;
    local_b = (double*)malloc(n * n * sizeof(double));
    DMI_read(b_addr, n * n * sizeof(double), local_b, DMI_INVALIDATE_READ, NULL);

```

のように、DMI_INVALIDATE_READ モードの DMI_read(...) を使うことが鍵になります。行列 B の場合、ここで DMI_read(...) したあとは再度 DMI_read(...) することがないため、その意味では DMI_GET_READ モードを使う方が自然ですが、データ転送を動的負荷分散させるためにはキャッシュを保持する DMI プロセスを増やす必要があるため、敢えて DMI_INVALIDATE_READ モードを使っています。

■ 3-5 補足

3-5-1 非同期 read/write に与えるローカルメモリを再利用して良いタイミング

非同期な DMI_write(int64_t addr, int64_t size, void *buf, ...) に関しては、その DMI_write(...) が返ってきた時点で、ローカルメモリ buf のデータは完全に処理系に渡されているため、すぐに buf を再利用したり解放しても問題ありません。すなわち、DMI_wait(...) によって完了を待たずとも、ローカルメモリ buf を再利用することができます。

非同期な DMI_read(int64_t addr, int64_t size, void *buf, ...) に関しては、この非同期操作が完了する直前でローカルメモリ buf にデータが格納されるわけなので、DMI_wait(...) によって完了を待たない限り、ローカルメモリ buf には意図するデータは格納されていません。当然、DMI_wait(...) によって完了を待つ前に、ローカルメモリ buf を再利用したり解放することはできません。

3-5-2 グローバル変数は使用できない

DMI では、グローバル変数を利用することはできません。何らか、DMI スレッド間でデータを共有する必要がある場合には、グローバルメモリを使用してください。

正確に言うと、DMI ではグローバル変数を記述することを文法上禁止しているわけではありませんが、おそらくプログラマが意図した動作は起きません。これは、DMI スレッドが内部的には pthread として実装されていることに起因します。つまり、 x というグローバル変数を記述したとき、この x は同一ノード

内に立ち上がっている DMI スレッドどうしても共有されていますが、別のノード内に立ち上がっている DMI スレッドどうしても共有されていません。言い換えると、`x` は DMI スレッドの数だけ存在するのではなく DMI プロセスの数だけ存在します。よって、グローバル変数を使っても、全 DMI スレッド間でのデータ共有が実現できるわけでもなく、各 DMI スレッド内に固有でどの関数からでも見えるようなデータを実現できるわけでもありません。たとえば、

```
int x = 0;

int64_t DMI_thread(int64_t addr)
{
    id = この DMI スレッドの id を取得;
    if(id == 0)
    {
        x = 1;
    }
    全 DMI スレッドで同期;
    printf("%d", x);
}
```

のようなプログラムを書いたときは、`id` が 0 の DMI スレッドと同じノード上に「たまたま」立ち上がっている DMI スレッドは 1 を出力しますが、残りの DMI スレッドは 0 を出力します。

3-5-3 スレッドローカルストレージは使用できない

gcc では、変数宣言の前に `__thread` を付けることで、スレッドローカルストレージを作ることができます。たとえば、

```
__thread int x = 0;

int64_t DMI_thread(int64_t addr)
{
    id = この DMI スレッドの id を取得;
    if(id == 0)
    {
        x = 1;
    }
    全 DMI スレッドで同期;
    printf("%d", x);
}
```

と記述することで、`id` が 0 の DMI スレッドだけが 1 を出力するようにできます。つまり、スレッドローカルストレージを使うことで、各 DMI スレッド内に固有でどの関数からでも見えるようなデータを実現できます。しかし、DMI は、スレッドを移動する際にスレッドローカルストレージ内のデータを移動しないため、スレッドが移動した途端に、スレッドローカルストレージ内のデータは意図しないものになります。よって、DMI ではスレッドローカルストレージは利用できません。ただし、スレッドの移動は DMI 1.4 からサポート予定の機能であるため、DMI 1.3 ではスレッドローカルストレージを利用してもプログラムは正しく動作します。

4 離散的な read/write

■ 4-1 概要

DMI_GET_READ モードの `DMI_read(...)` や DMI_PUT_WRITE モードの `DMI_write(...)` を使うと get/put 操作を実現できますが、これらは、ある連続したグローバルメモリアドレス領域に対してしか発行することができません。これに対して、ここで説明する離散的な read/write を使うと、離散的なグローバルメモリアドレス領域に対して一括して get/put 操作を発行することができます。たとえば、Fig.??のように 4 個のグローバルメモリを考え、各グローバルメモリは 2 個のページから構成されているとします。このとき、Fig.??に示すような 8 箇所の離散的なグローバルメモリアドレス領域の値を、一括して効率的に get/put することができます。この離散的な read/write は、put/get 操作の最も汎用的な形態といえます。

有限要素法による応力解析や流体解析などの領域分割型の並列科学技術計算などでは、各反復計算において隣接する領域の境界部分の値だけを get する必要があるため、この離散的な read/write が効果を発揮すると思われます。

■ 4-2 API

離散的な read/write は、以下の 4 つの API で実現できます。まず `DMI_group_init(...)` でローカル不透明オブジェクト `group` を初期化し、そのグループを使って `DMI_group_read(...)/DMI_group_write(...)` を行い、最後に `DMI_group_destroy(...)` でグループを破棄します。なお、ローカル不透明オブジェクトに関しては??で詳しく説明します。離散的な read/write においては、不透明オブジェクトに関する面倒な扱いは不要であるため、API を利用するための情報を隠蔽してまとめたものという程度に理解していれば十分です。

- ```
int32_t DMI_group_init(DMI_local_group_t *group, int64_t *addrs,
 int64_t *sizes, int64_t *ptr_offsets, int32_t group_num);
```

離散的にアクセスするグローバルメモリアドレス領域を定義して、その結果をローカル不透明オブジェクトとして `group` に返します。グローバルメモリアドレス領域は、要素数が `group_num` の配列 `addr_offsets`, `ptr_offsets`, `sizes` によって定義します。ここで、`group_num` 個の離散的なグローバルメモリアドレス領域を定義するとき、 $i$  番目のグローバルメモリアドレス領域の先頭アドレスを `addrs[i]` に、 $i$  番目のグローバルメモリアドレス領域のサイズを `sizes[i]` に指定します。また、 $i$  番目のグローバルメモリアドレス領域を `DMI_group_read(...)/DMI_group_write(...)` によって get/put するときに、`DMI_group_read(...)/DMI_group_write(...)` の引数に与えるローカルメモリアドレスを基準として、どのオフセットの位置からデータを get/put するかを `ptr_offsets[i]` に指定します。
- ```
int32_t DMI_group_destroy(DMI_local_group_t *group);
```

ローカル不透明オブジェクトの `group` を破棄します。
- ```
int32_t DMI_group_read(DMI_group_group_t *group, void *in_ptr,
```

```
DMI_local_status_t *status);:
```

ローカル不透明オブジェクト `group` に定義された離散的なグローバルメモリアドレス領域のデータをローカルメモリ `in_ptr` に read します。`i` 番目のグローバルメモリアドレス領域 `[addr[i], addr[i]+size[i])` のデータは、`[in_ptr[ptr_offsets[i]], in_ptr[ptr_offsets[i]]+size[i])` に格納されます。

- `int32_t DMI_group_write(DMI_local_group_t *group, void *out_ptr, DMI_local_status_t *status);:`

ローカル不透明オブジェクト `group` に定義された離散的なグローバルメモリアドレス領域に対して、ローカルメモリ `out_ptr` からデータを write します。`i` 番目のグローバルメモリアドレス領域 `[addr[i], addr[i]+size[i])` には、`[out_ptr[ptr_offsets[i]], out_ptr[ptr_offsets[i]]+size[i])` のデータが書き込まれます。

### ■ 4-3 具体例

Fig.??に示すように、8箇所の離散的なグローバルメモリアドレス領域の値を、一括してローカルメモリに get/put する場合を考えます。このときは、以下のようにプログラムを記述します：

各配列の中の値がどう対応しているかを Fig.??と見比べてみてください。

### ■ 4-4 補足

#### 4-4-1 離散的な read/write の性能

離散的な read/write は、単純に、各グローバルメモリアドレス領域に対して get/put を繰り返すわけではなく、より効率的に動作します。Fig.??に示すような 8 箇所の離散的なグローバルメモリアドレス領域を `DMI_group_init(...)` で定義したとき、これらの 8 箇所の離散的なグローバルメモリアドレス領域が、ページごとに整理されて、その情報が不透明オブジェクト `group` に書き込まれます。つまり、Fig.??の場合には、4 つの領域に整理され、`DMI_group_read(...)/DMI_group_write(...)` が実際の get/put 操作を行うときには、8 回のページフォルトではなく、4 回のページフォルトが引き起こされ、4 個のページフォルト通知だけが各ページのオーナーへと通知されます。このように、 $x$  個のページにまたがる  $y$  個の離散的なグローバルメモリアドレス領域を指定した場合、 $y$  がいくら大きくとも、内部的には最大  $x$  回のページフォルトしか発生せず、よってキャッシュ管理も最大  $x$  回しか発生しません。

なお、不透明オブジェクトを生成する `DMI_group_init(...)` と、実際に get/put を行う `DMI_group_read(...)/DMI_group_write(...)` を分離しているのは、領域分割型の並列科学技術計算における反復計算では、同一の離散的なグローバルメモリアドレス領域に対して、何度も get/put するためです。つまり、`DMI_group_init(...)` で作ったローカル不透明オブジェクトを使って、何度も get/put することができるようにしています。


## 5 read-write-set

---

執筆中 ...

### ■ 5-1 ローカル不透明オブジェクトとグローバル不透明オブジェクト

---



# Chap.5 同期

DMI では pthread プログラミングとほとんど同様の方法で同期を実現することができます。pthread プログラミングにおいて最も基本的な同期の手段は、排他制御変数 (mutex) と条件変数 (cond) だと思われるため、本節では、まず排他制御変数と条件変数について解説します。そのあと、バリア操作について説明し、より柔軟で高度な同期を実現するための手段として、組み込みの read-modify-write やユーザ定義の read-modify-write について説明します。

## 1 排他制御変数

### ■ 1-1 API

API は以下のとおりです。セマンティクスは pthread の排他制御変数の API と同様のため、pthread のマニュアルを参照してください：

- `int32_t DMI_mutex_init(int64_t mutex_addr);`  
mutex\_addr で示されるグローバルメモリアドレスを、排他制御変数として初期化します。  
mutex\_addr は DMI\_mutex\_t のサイズを持つグローバルメモリである必要があります。
- `int32_t DMI_mutex_destroy(int64_t mutex_addr);`  
mutex\_addr で示されるグローバルメモリアドレスに作られている排他制御変数を破棄します。
- `int32_t DMI_mutex_lock(int64_t mutex_addr);`  
排他制御変数をロックします。
- `int32_t DMI_mutex_unlock(int64_t mutex_addr);`  
排他制御変数をアンロックします。
- `int32_t DMI_mutex_trylock(int64_t mutex_addr, int32_t *flag_ptr);`  
排他制御変数のロックを試みます。ロックに成功した場合、排他制御変数はロックされ、\*flag\_ptr に DMI\_TRUE が格納されます。すでにこの排他制御変数がロックされており、ロックに失敗した場合には、\*flag\_ptr に DMI\_FALSE が格納されます。

## ■ 1-2 具体例

1 個のカウント変数を、複数の DMI スレッドが排他制御しながらインクリメントする処理は以下のよう  
に記述できます：

```
#include "dmi_api.h"

typedef struct targ_t
{
 int32_t rank; /* DMI スレッドのランク */
 int32_t iter_num; /* カウント変数をカウントする回数 */
 int64_t mutex_addr; /* 排他制御変数 */
 int64_t counter_addr; /* カウント変数 */
}targ_t;

void DMI_main(int argc, char **argv)
{
 DMI_node_t node;
 DMI_node_t *nodes;
 DMI_thread_t *threads;
 targ_t targ;
 int32_t i, j, value, rank, node_num, thread_num, iter_num;
 int64_t targ_addr, mutex_addr, counter_addr;

 if(argc != 4)
 {
 fprintf(stderr, "usage : %s node_num thread_num iter_num\n", argv[0]);
 exit(1);
 }
 node_num = atoi(argv[1]);
 thread_num = atoi(argv[2]);
 iter_num = atoi(argv[3]);
 nodes = (DMI_node_t*)malloc(node_num * sizeof(DMI_node_t));
 threads = (DMI_thread_t*)malloc(node_num * thread_num * sizeof(DMI_thread_t));
 DMI_mmap(&targ_addr, sizeof(targ_t), node_num * thread_num, NULL);
 DMI_mmap(&mutex_addr, sizeof(DMI_mutex_t), 1, NULL); /* 排他制御変数のグローバルメモリを確保 */
 DMI_mmap(&counter_addr, sizeof(int32_t), 1, NULL); /* カウント変数のグローバルメモリを確保 */
 DMI_mutex_init(mutex_addr); /* 排他制御変数を初期化 */
 value = 0;
 DMI_write(counter_addr, sizeof(int32_t), &value, DMI_EXCLUSIVE_WRITE, NULL); /* カウンタ
変数の値を 0 に初期化 */
 for(i = 0; i < node_num * thread_num; i++)
 {
 targ.rank = i;
 targ.iter_num = iter_num;
 targ.mutex_addr = mutex_addr; /* 排他制御変数のグローバルアドレス */
 targ.counter_addr = counter_addr; /* カウント変数のグローバルアドレス */
 DMI_write(targ_addr + i * sizeof(targ_t), sizeof(targ_t), &targ, DMI_EXCLUSIVE_WRITE, NULL)
 }
 for(i = 0; i < node_num; i++)
 {
 DMI_poll(&node);
 if(node.state == DMI_OPEN)
 {
```



```

 DMI_welcome(node.dmi_id);
 nodes[i] = node;
 }
}
rank = 0;
for(i = 0; i < node_num; i++)
{
 node = nodes[i];
 for(j = 0; j < thread_num; j++)
 {
 DMI_create(&threads[rank], node.dmi_id, targ_addr + rank * sizeof(targ_t), NULL);
 rank++;
 }
}
for(rank = 0; rank < node_num * thread_num; rank++)
{
 DMI_join(threads[rank], NULL, NULL);
}
DMI_read(targ.counter_addr, sizeof(int32_t), &value, DMI_INVALIDATE_READ, NULL); /* 最
後にカウンタ変数の値を読む */
printf("total : %d\n", value);
for(i = 0; i < node_num; i++)
{
 DMI_poll(&node);
 if(node.state == DMI_CLOSE)
 {
 DMI_goodbye(node.dmi_id);
 }
}
DMI_mutex_destroy(mutex_addr); /* 排他制御変数を破棄 */
DMI_munmap(mutex_addr, NULL); /* 排他制御変数のグローバルメモリを解放 */
DMI_munmap(counter_addr, NULL); /* カウンタ変数のグローバルメモリを解放 */
DMI_munmap(targ_addr, NULL);
free(threads);
free(nodes);
return;
}

int64_t DMI_thread(int64_t addr)
{
 targ_t targ;
 int32_t i, value;

 DMI_read(addr, sizeof(targ_t), &targ, DMI_GET_READ, NULL);
 for(i = 0; i < targ.iter_num; i++) /* iter_num 回だけカウンタ変数をインクリメント */
 {
 DMI_mutex_lock(targ.mutex_addr); /* 排他制御変数をロック */
 DMI_read(targ.counter_addr, sizeof(int32_t), &value, DMI_GET_READ, NULL); /* カウン
タ変数の値を読む */
 value++; /* インクリメント */
 DMI_write(targ.counter_addr, sizeof(int32_t), &value, DMI_PUT_WRITE, NULL); /* カウ
ンタ変数の値を書く */
 DMI_mutex_unlock(targ.mutex_addr); /* 排他制御変数をアンロック */
 }
 return DMI_NULL;
}

```

```
}
```

## 2 条件変数

### ■ 2-1 API

API は以下のとおりです。セマンティクスは pthread の条件変数の API と同様のため、pthread のマニュアルを参照してください：

- `int32_t DMI_cond_init(int64_t cond_addr);` :  
cond\_addr で示されるグローバルメモリアドレスを、条件変数として初期化します。cond\_addr は DMI\_cond\_t のサイズを持つグローバルメモリである必要があります。
- `int32_t DMI_cond_destroy(int64_t cond_addr);` :  
条件変数を破棄します。
- `int32_t DMI_cond_wait(int64_t cond_addr, int64_t mutex_addr);` :  
排他制御変数と組み合わせて、wait します。
- `int32_t DMI_cond_signal(int64_t cond_addr);` :  
wait している DMI スレッドのうち、どれか 1 個の DMI スレッドを起こします。
- `int32_t DMI_cond_broadcast(int64_t cond_addr);` :  
wait している DMI スレッドすべてを起こします。

### ■ 2-2 具体例

排他制御変数と条件変数を組み合わせることで、DMI スレッドすべてでバリア操作を行うプログラムは、以下のように記述できます：

```
#include "dmi_api.h"

typedef struct targ_t
{
 int32_t rank; /* DMI スレッドのランク */
 int64_t sync_addr; /* 排他制御変数と条件変数とカウンタ変数の組み合わせ */
 int32_t pnum; /* DMI スレッドの個数 */
}targ_t;

typedef struct sync_t
{
 DMI_mutex_t mutex; /* 排他制御変数 */
 DMI_cond_t cond; /* 条件変数 */
 int64_t counter; /* カウンタ変数（その時点でいくつの DMI スレッドがバリア操作に到達しているか） */
}sync_t;

void barrier(int64_t sync_addr, int32_t pnum);
```

```

void DMI_main(int argc, char **argv)
{
 DMI_node_t node;
 DMI_node_t *nodes;
 DMI_thread_t *threads;
 targ_t targ;
 int32_t i, j, value, rank, node_num, thread_num;
 int64_t targ_addr, sync_addr;

 if(argc != 3)
 {
 fprintf(stderr, "usage : %s node_num thread_num\n", argv[0]);
 exit(1);
 }
 node_num = atoi(argv[1]);
 thread_num = atoi(argv[2]);
 nodes = (DMI_node_t*)malloc(node_num * sizeof(DMI_node_t));
 threads = (DMI_thread_t*)malloc(node_num * thread_num * sizeof(DMI_thread_t));
 DMI_mmap(&targ_addr, sizeof(targ_t), node_num * thread_num, NULL);
 DMI_mmap(&sync_addr, sizeof(sync_t), 1, NULL); /* バリア操作のためのグローバルメモリを確保 */
 DMI_mutex_init((int64_t)&((sync_t*)sync_addr->mutex)); /* 排他制御変数を初期化 */
 DMI_cond_init((int64_t)&((sync_t*)sync_addr->cond)); /* 条件変数を初期化 */
 value = 0;
 DMI_write((int64_t)&((sync_t*)sync_addr->counter), sizeof(int32_t), &value, DMI_EXCLUSIVE_WRITE,
 ウンタ変数の値を 0 に初期化 */
 for(i = 0; i < node_num * thread_num; i++)
 {
 targ.rank = i;
 targ.sync_addr = sync_addr; /* バリア操作のためのグローバルアドレス */
 targ.pnum = node_num * thread_num;
 DMI_write(targ_addr + i * sizeof(targ_t), sizeof(targ_t), &targ, DMI_EXCLUSIVE_WRITE, NULL)
 }
 for(i = 0; i < node_num; i++)
 {
 DMI_poll(&node);
 if(node.state == DMI_OPEN)
 {
 DMI_welcome(node.dmi_id);
 nodes[i] = node;
 }
 }
 rank = 0;
 for(i = 0; i < node_num; i++)
 {
 node = nodes[i];
 for(j = 0; j < thread_num; j++)
 {
 DMI_create(&threads[rank], node.dmi_id, targ_addr + rank * sizeof(targ_t), NULL);
 rank++;
 }
 }
 for(rank = 0; rank < node_num * thread_num; rank++)
 {
 DMI_join(threads[rank], NULL, NULL);
 }
}

```

```

 }
 for(i = 0; i < node_num; i++)
 {
 DMI_poll(&node);
 if(node.state == DMI_CLOSE)
 {
 DMI_goodbye(node.dmi_id);
 }
 }
 DMI_cond_destroy((int64_t)&((sync_t*)sync_addr->cond)); /* 条件変数を破棄 */
 DMI_mutex_destroy((int64_t)&((sync_t*)sync_addr->mutex)); /* 排他制御変数を破棄 */
 DMI_munmap(sync_addr, NULL); /* バリア操作のためのグローバルメモリを解放 */
 DMI_munmap(targ_addr, NULL);
 free(threads);
 free(nodes);
 return;
}

int64_t DMI_thread(int64_t addr)
{
 targ_t targ;
 int32_t i, value;

 DMI_read(addr, sizeof(targ_t), &targ, DMI_GET_READ, NULL);
 for(i = 0; i < 20; i++) /* バリア操作を 20 回繰り返す */
 {
 barrier(targ.sync_addr, targ.pnum); /* バリア操作 */
 }
 return DMI_NULL;
}

void barrier(int64_t sync_addr, int32_t pnum)
{
 int value;

 DMI_mutex_lock((int64_t)&((sync_t*)sync_addr->mutex)); /* 排他制御変数をロック */
 DMI_read((int64_t)&((sync_t*)sync_addr->counter), sizeof(int32_t), &value, DMI_GET_READ, NULL);
 ウンタ変数の値を読む /*
 value++; /* カウンタ変数をインクリメント */
 if(value < pnum) /* 「最後」の DMI スレッドでなければ */
 {
 DMI_write((int64_t)&((sync_t*)sync_addr->counter), sizeof(int32_t), &value, DMI_PUT_WRITE);
 ウンタ変数の値を書く /*
 DMI_cond_wait((int64_t)&((sync_t*)sync_addr->cond), (int64_t)&((sync_t*)sync_addr->mutex));
 条件変数の wait /*
 }
 else /* 「最後」の DMI スレッドでならば */
 {
 value = 0;
 DMI_write((int64_t)&((sync_t*)sync_addr->counter), sizeof(int32_t), &value, DMI_PUT_WRITE);
 ウンタ変数の値を書く /*
 DMI_cond_broadcast((int64_t)&((sync_t*)sync_addr->cond)); /* 条件変数の
 broadcast /*
 }
 DMI_mutex_unlock((int64_t)&((sync_t*)sync_addr->mutex)); /* 排他制御変数をアンロック */

```

```
return;
}
```

上記の DMI プログラムを実行すると、

## 3 バリア操作

### ■ 3-1 API

??で示したように、排他制御変数と条件変数を組み合わせることでバリア操作を実現できますが、排他制御変数と条件変数を利用するバリア操作は遅いです。これは共有メモリ環境上の pthread プログラミングにもいえることで、本当に高性能なバリア操作を実現する場合、より低レベルの同期プリミティブやビジーウェイトを組み合わせる必要があります。そこで、DMI では、より高性能なバリア操作を API としてまとめて提供しています。このバリア操作は、単にバリアを行うだけでなく、おまけの機能として double 型の変数の加算操作も行えるようになっています。

API は以下のとおりです。ローカル不透明オブジェクトとグローバル不透明オブジェクトに関しては??節を参照してください：

- `int32_t DMI_barrier_init(int64_t barrier_addr);`  
barrier\_addr で示されるグローバルメモリアドレスを、バリア操作のためのグローバル不透明オブジェクトとして初期化します。barrier\_addr は DMI\_barrier\_t のサイズを持つグローバルメモリである必要があります。
- `int32_t DMI_barrier_destroy(int64_t barrier_addr);`  
バリア操作のためのグローバル不透明オブジェクトを破棄します。
- `int32_t DMI_local_barrier_init(DMI_local_barrier_t *barrier, int64_t barrier_addr);`  
バリア操作のためのグローバル不透明オブジェクト barrier\_addr を与えることで、実際に各 DMI スレッドがバリア操作に使うためのローカル不透明オブジェクト barrier を初期化します。
- `int32_t DMI_local_barrier_destroy(DMI_local_barrier_t *barrier);`  
ローカル不透明オブジェクト barrier を破棄します。
- `int32_t DMI_local_barrier_allreduce(DMI_local_barrier_t *barrier, double sub_sum, double *sum_ptr, int32_t pnum);`  
ローカル不透明オブジェクト barrier を利用して、pnum 個の DMI スレッドの間で Allreduce 操作（つまりバリア操作）を行います。各 DMI スレッドの DMI\_local\_barrier\_allreduce(...) は、全 DMI スレッドが DMI\_local\_barrier\_allreduce(...) を呼び出した時点で、全 DMI スレッドが与えた sub\_sum の総和を sum\_ptr に格納したうえで返ります。これにより、double 型の値の Allreduce 操作（つまりバリア操作）を実現できます。バリア操作を行う際には、全 DMI スレッドが pnum として同一の値を指定しなければなりません。

## ■ 3-2 具体例

??節のプログラムを、このバリア操作の API を使って記述すると以下のようになります。排他制御変数と条件変数を使うよりもこちらの方が高速です：

```
#include "dmi_api.h"

typedef struct targ_t
{
 int32_t rank; /* DMI スレッドのランク */
 int64_t barrier_addr; /* グローバル不透明オブジェクト */
 int32_t pnum; /* DMI スレッドの個数 */
}targ_t;

void DMI_main(int argc, char **argv)
{
 DMI_node_t node;
 DMI_node_t *nodes;
 DMI_thread_t *threads;
 targ_t targ;
 int32_t i, j, value, rank, node_num, thread_num;
 int64_t targ_addr, barrier_addr;

 if(argc != 3)
 {
 fprintf(stderr, "usage : %s node_num thread_num\n", argv[0]);
 exit(1);
 }
 node_num = atoi(argv[1]);
 thread_num = atoi(argv[2]);
 nodes = (DMI_node_t*)malloc(node_num * sizeof(DMI_node_t));
 threads = (DMI_thread_t*)malloc(node_num * thread_num * sizeof(DMI_thread_t));
 DMI_mmap(&targ_addr, sizeof(targ_t), node_num * thread_num, NULL);
 DMI_mmap(&barrier_addr, sizeof(barrier_t), 1, NULL); /* グローバル不透明オブジェクトのグローバルメモリを確保 */
 DMI_barrier_init(barrier_addr); /* グローバル不透明オブジェクトの初期化 */
 for(i = 0; i < node_num * thread_num; i++)
 {
 targ.rank = i;
 targ.barrier_addr = barrier_addr; /* グローバル不透明オブジェクトのグローバルアドレス */
 targ.pnum = node_num * thread_num;
 DMI_write(targ_addr + i * sizeof(targ_t), sizeof(targ_t), &targ, DMI_EXCLUSIVE_WRITE, NULL);
 }
 for(i = 0; i < node_num; i++)
 {
 DMI_poll(&node);
 if(node.state == DMI_OPEN)
 {
 DMI_welcome(node.dmi_id);
 nodes[i] = node;
 }
 }
 rank = 0;
 for(i = 0; i < node_num; i++)
```

```

{
 node = nodes[i];
 for(j = 0; j < thread_num; j++)
 {
 DMI_create(&threads[rank], node.dmi_id, targ_addr + rank * sizeof(targ_t), NULL);
 rank++;
 }
}
for(rank = 0; rank < node_num * thread_num; rank++)
{
 DMI_join(threads[rank], NULL, NULL);
}
for(i = 0; i < node_num; i++)
{
 DMI_poll(&node);
 if(node.state == DMI_CLOSE)
 {
 DMI_goodbye(node.dmi_id);
 }
}
DMI_barrier_destroy(barrier_addr); /* グローバル不透明オブジェクトの破棄 */
DMI_munmap(barrier_addr, NULL);
DMI_munmap(targ_addr, NULL);
free(threads);
free(nodes);
return;
}

int64_t DMI_thread(int64_t addr)
{
 targ_t targ;
 int32_t i, value;
 double dummy;
 DMI_local_barrier_t local_barrier; /* ローカル不透明オブジェクト */

 DMI_read(addr, sizeof(targ_t), &targ, DMI_GET_READ, NULL);
 DMI_local_barrier_init(&local_barrier, targ.barrier_addr); /* ローカル不透明オブジェクトの初期
化 */
 for(i = 0; i < 20; i++)
 {
 DMI_local_barrier_allreduce(&local_barrier, 0, &dummy, targ.pnum); /* バリア操作 */
 }
 DMI_local_barrier_destroy(&local_barrier); /* ローカル不透明オブジェクトの破棄 */
 return DMI_NULL;
}

```

### ■ 3-3 補足

#### 3-3-1 DMI\_local\_barrier\_allreduce(...) が返るタイミング

DMI\_local\_barrier\_allreduce(...) が返るタイミングを正確に説明します。ある DMI スレッド  $t$  が、グローバル不透明オブジェクト `barrier_addr` を基に生成されたローカル不透明オブジェク

ト barrier を利用して `DMI_local_barrier_allreduce(...)` を呼び出すとします。一般には、多数の DMI スレッドがこのグローバル不透明オブジェクト `barrier_addr` を基にローカル不透明オブジェクトを生成しています。このとき、DMI スレッド  $t$  の `DMI_local_barrier_allreduce(...)` が返るのは、同一のグローバル不透明オブジェクト `barrier_addr` から生成されたローカル不透明オブジェクトを利用した `DMI_local_barrier_allreduce(...)` が合計 `pnum` 回呼び出された時点です。

## 4 スレッドスケジューリング

### ■ 4-1 API

DMI スレッドを眠らせたり、特定のスレッドを起こしたりすることができます。この API を使うと、より細粒度な同期を実現できます。API は以下のとおりです：

- `int32_t DMI_suspend(void);`

この `DMI_suspend(...)` を呼び出した DMI スレッドを眠らせます。`DMI_wake(...)` によって起こされるまで、この DMI スレッドは眠り続けます。

- `int32_t DMI_wake(DMI_thread_t dmi_thread, DMI_local_status_t *status);`

スレッドハンドル `dmi_thread` で指定される DMI スレッドを起こします。`status` を指定することで非同期操作にできます。

## 5 組み込みの read-modify-write

### ■ 5-1 read-modify-write

#### 5-1-1 read-modify-write

本節では、read-modify-write とは何かを説明するために、DMI の話を離れて、通常の CPU と共有メモリ環境に関する話をします。

まず、そもそも共有メモリ環境上の排他制御変数 (`mutex`) のロック操作がどのように実装されているかを考えてみます。おそらく、現在その排他制御変数をロックしているかどうかを管理するための変数 `flag` があって、

```
void mutex_lock(mutex_t *mutex)
{
 while(1)
 {
 while(mutex->flag == 1); /* ロックが解除されるまでビジーウェイト */
 if(mutex->flag == 0) /* ロックされていないならば */
```



```
 {
 mutex->flag = 1; /* ロックする */
 return;
 }
 return;
}
```

のような実装になっているのではないかと考えるわけですが，これでは排他制御は失敗します．たとえば，あるスレッド  $t_1$  が `mutex->flag == 0` であることを確認し，`mutex->flag` を 1 に書き換えてロックを成功させたとします．このとき，スレッド  $t_1$  が `mutex->flag == 0` であることを確認してから `mutex->flag` を 1 に書き換えるまでの間は `mutex->flag` の値はまだ 0 のままです．よって，この間に，別のスレッド  $t_2$  が `mutex->flag == 0` であることを確認して，スレッド  $t_2$  も `mutex->flag` を 1 に書き換えてロックを成功させてしまう可能性があります．つまり，複数のスレッドがロックを成功させたと誤ってしまう可能性があり，これでは排他制御になっていません．この問題の根源は，CPU が各 read/write の単位でしかアトミックな実行を保証してくれない（と今は仮定している）ために，`mutex->flag == 0` であることを確認してから `mutex->flag` を 1 に書き換えるまでに「間」が存在してしまうことに起因しています．そこで，通常の CPU では，(1) あるアドレスの値を読み出し，(2) その値に対して何らかの操作を行い，(3) そのアドレスに何らかの値を書き込む，という操作をアトミックに実行するための命令をいくつか提供しています<sup>\*1</sup>．これが read-modify-write と呼ばれる命令群であり，代表的な read-modify-write としては，compare-and-swap，fetch-and-store，test-and-set などがあります．これらの read-modify-write を使うと，今説明したような排他制御変数（mutex）などを，read-modify-write がない場合と比較して簡単かつ効率的に実装できます．また，データ構造によっては，そのデータ構造に対する read/write を排他制御変数（mutex）で排他制御するのではなく，read-modify-write を上手に使うことでより効率的な排他制御が行うことができます．詳細が知りたい方は，lock-free，wait-free のキーワードで調べてください．以下では，compare-and-swap と fetch-and-store が具体的にどのような命令なのかを説明します．

### 5-1-2 compare-and-swap

compare-and-swap は次の関数をアトミックに実行する命令です：

```
int compare_and_swap(int *p, int x, int y)
{
 if(*p == x)
 {
 *p = y;
 return 1;
 }
 return 0;
}
```

<sup>\*1</sup> 実は，非常にトリッキーなアルゴリズムを使うと，CPU が各 read/write の単位でしかアトミックな実行を保証してくれないという仮定の下でも排他制御を実現することはできますが，read-modify-write を使う場合の排他制御と比較すると効率は悪いです．

つまり、指定したメモリアドレス  $p$  の値が  $x$  と等しければ、メモリアドレス  $p$  の値を  $y$  に書き換えて、成功を表す 1 を返します。指定したメモリアドレス  $p$  の値が  $x$  と等しくなければ、何も行わずに 0 を返します。以上の操作をアトミックに実行します。

たとえば、上記で例に挙げた排他制御変数のロック操作は、compare-and-swap を使うことで以下のよう実装できます：

```
void mutex_lock(mutex_t *mutex)
{
 while(1)
 {
 while(mutex->flag == 1); /* ロックが解除されるまでビジーウェイト */
 int ret = compare_and_swap(&mutex->flag, 0, 1);
 if(ret == 1) return; /* ロック成功 */
 }
 return;
}
```

### 5-1-3 fetch-and-store

fetch-and-store は次の関数をアトミックに実行する命令です：

```
void fetch_and_store(int *p, int in, int *out)
{
 *out = *p;
 *p = in;
 return;
}
```

つまり、指定したメモリアドレス  $p$  のその時点での値を  $out$  に格納したあとで、メモリアドレス  $p$  の値を  $in$  の値に書き換えるという作業をアトミックに行います。

## ■ 5-2 API

DMI の話に戻ります。DMI では、compare-and-swap と fetch-and-store を行うための以下のような API を提供しています：

- `int32_t DMI_cas(int64_t addr, int64_t size, void *cmp_ptr, void *swap_ptr, int8_t *flag_ptr, int8_t mode, DMI_local_status_t *status);`

compare-and-swap を行います。グローバルメモリアドレス  $addr$  から  $size$  バイトのデータを、ローカルメモリアドレス  $cmp\_ptr$  から  $size$  バイトのデータと比較し、完全に一致していれば、グローバルメモリアドレス  $addr$  から  $size$  バイトのデータをローカルメモリアドレス  $cmp\_ptr$  から  $size$  バイトのデータに置き換えて、 $flag\_ptr$  に `DMI_TRUE` を格納してから返ります。1 バイトでも一致していなければ、グローバルメモリには変更を行わず、 $flag\_ptr$  に `DMI_FALSE` を格納してから返ります。以上の操作をアトミックに行います。mode としては、`DMI_EXCLUSIVE_ATOMIC`、

DMI\_PUT\_ATOMIC の 2 種類のモードを指定できます．なお，グローバルメモリアドレス領域 [addr,addr+size) は，必ず 1 個のページ内に収まっている必要があります．

- `int32_t DMI_fas(int64_t addr, int64_t size, void *out_ptr, void *in_ptr, int8_t mode, DMI_local_status_t *status);`

fetch-and-store を行います．その時点におけるグローバルメモリアドレス addr から size バイトのデータを，ローカルメモリアドレス in\_ptr に格納したあとで，グローバルメモリアドレス addr から size バイトのデータをローカルメモリアドレス cmp\_ptr から size バイトのデータに置き換えます．以上の操作をアトミックに行います．mode としては，DMI\_EXCLUSIVE\_ATOMIC, DMI\_PUT\_ATOMIC の 2 種類のモードを指定できます．なお，グローバルメモリアドレス領域 [addr,addr+size) は，必ず 1 個のページ内に収まっている必要があります．

DMI\_EXCLUSIVE\_ATOMIC, DMI\_PUT\_ATOMIC の各モードは，この read-modify-write に伴うキャッシュ管理を明示的に指示するためのものです．キャッシュ管理の動作は，それぞれ，DMI\_EXCLUSIVE\_WRITE, DMI\_PUT\_WRITE と同様です．

## 6 ユーザ定義の read-modify-write

### ■ 6-1 概要

DMI\_cas(...) と DMI\_fas(...) を使えば，compare-and-swap と fetch-and-store を実現できますが，逆に言えば，compare-and-swap と fetch-and-store しか実現できません．そこで DMI では，組み込みの compare-and-swap と fetch-and-store に限らず，プログラマが任意の read-modify-write を定義するための機能を提供しています．

当然，多様な read-modify-write が提供されていればいるほど，意図する同期を効率的に実装することができます．たとえば，カウンタ変数 counter を複数の DMI スレッドでインクリメントする処理を考えます．read-modify-write として compare-and-swap と fetch-and-store しか使えないのであれば，これらの read-modify-write や通常の read/write などを組み合わせて，カウンタ変数 counter のインクリメントをアトミックに実現する必要があります．排他制御変数を使って排他する場合でも，結局のところ内部的には，read-modify-write や通常の read/write が組み合わせられている点に注意してください．これに対して，仮に「カウンタ変数 counter をアトミックにインクリメントする」という read-modify-write をプログラマが新たに作ることができれば，その read-modify-write 一命令だけでカウンタ変数 counter のインクリメントを実現できるため，高速です．

このように，プログラマが任意の read-modify-write を作り出すことで，意図する同期をより高速に実装することができます．

## ■ 6-2 API

### 6-2-1 簡単化した説明

DMI におけるユーザ定義の read-modify-write の API は、プログラマに最大限の柔軟性を提供しようとした結果かなり複雑なものになっているので、順を追って解説します。

まず、プログラマは以下のようなプロトタイプを持った `DMI_function(...)` という名前の関数を定義し、この `DMI_function(...)` の中に、read-modify-write の命令を記述します。関数名は必ず `DMI_function(...)` である必要があります：

```
• void DMI_function(void *page_ptr, int64_t size, void *out1_ptr,
 int64_t out1_size, void *out2_ptr, int64_t out2_size, void *in_ptr,
 int64_t in_size, int8_t tag) { ... }
```

そのうえで、`DMI_atomic(...)` を呼び出すと、`DMI_atomic(...)` に与えた引数が `DMI_function(...)` に渡されて実行され、ユーザ定義の read-modify-write が実行されます。`DMI_atomic(...)` の API は以下のとおりです：

```
• int32_t DMI_atomic(int64_t addr, int64_t size, void *out1_ptr, int64_t
 out1_size, void *out2_ptr, int64_t out2_size, void *in_ptr, int64_t
 in_size, int8_t tag, int8_t mode, status_t *status);
```

要するに、概略としては、プログラマが `DMI_function(...)` という関数の中に任意の read-modify-write を記述したうえで、`DMI_atomic(...)` を呼び出すと、`DMI_atomic(...)` に与えた引数が `DMI_function(...)` に渡されて実行され、意図した read-modify-write を実行できるという仕組みです。特に `DMI_atomic(...)` に渡す `tag` の値がそのまま `DMI_function(...)` の `tag` に渡されるので、`DMI_function(...)` の中ではこの `tag` の値に基づいて、どの read-modify-write を実行するかを振り分けることができます。よって、プログラムの概形は以下のようになります：

```
void DMI_function(void *page_ptr, int64_t size, void *out1_ptr, int64_t out1_size, void *out2_ptr,
{
 switch(tag)
 {
 case 12345:
 ...; /* read-modify-write その1 */
 break;
 case 12346:
 ...; /* read-modify-write その2 */
 break;
 case 12347:
 ...; /* read-modify-write その3 */
 break;
 ...;
 }
 return;
}
```

```
int64_t DMI_thread(int64_t addr)
{
 ...;
 DMI_atomic(addr, size, out1_ptr, out1_size, out2_ptr, out2_size, in_ptr, in_size, tag, mode, st
 ...;
}
```

### 6-2-2 正確な説明

詳しいセマンティクスについて説明します。DMI\_atomic(...) に渡す引数と、それによって呼び出された DMI\_function(...) に渡される引数の関係は以下のように決まります：

- DMI\_atomic(...) に渡す addr はグローバルメモリアドレスです。グローバルメモリアドレス領域 [addr,addr+size) は 1 ページに収まる領域でなければなりません。このとき、グローバルメモリアドレス領域 [addr,addr+size) に対応するページの実体のメモリアドレスが、DMI\_function(...) の引数 page\_ptr として渡されます。より実装レベルの表現で言えば、DMI\_function(...) は、グローバルメモリアドレス領域 [addr,addr+size) が属するページのオーナーの DMI プロセスで呼び出されます。そして、オーナーのメモリプールの中でそのグローバルメモリアドレス領域を格納しているメモリアドレスが page\_ptr に渡されます。
- DMI\_atomic(...) に渡す out1\_ptr と out2\_ptr はローカルメモリアドレスです。DMI\_atomic(...) に渡した out1\_ptr から out1\_size バイトのデータと out2\_ptr から out2\_size バイトのデータが、それぞれ、DMI\_function(...) の out1\_ptr と out2\_ptr にそのまま渡されます。また、DMI\_function(...) の out1\_size と out2\_size には、それぞれ、DMI\_atomic(...) に渡した out1\_size と out2\_size が渡されます。
- DMI\_atomic(...) に渡した tag の値は、DMI\_function(...) の tag に渡されます。
- DMI\_atomic(...) に渡した in\_size の値は、DMI\_function(...) の in\_size に渡されます。また、DMI\_function(...) に渡される in\_ptr は in\_size バイトのサイズのローカルメモリであり、初期的には何もデータが格納されているわけではなく、DMI\_function(...) が任意のデータを格納するために利用します。すると、DMI\_function(...) が終了したときに、DMI\_function(...) の in\_ptr から in\_size バイトのデータが、この DMI\_function(...) を呼び出した DMI\_atomic(...) の in\_ptr にそのまま渡されます。つまり、この in\_ptr は DMI\_function(...) の「返り値」です。よって、DMI\_atomic(...) に渡す in\_ptr は、少なくとも in\_size バイトのサイズを持ったローカルメモリである必要があります。

以上の関係を Fig.?? にまとめます。要するに、ポイントをまとめると以下のとおりです：

- out1\_ptr と out2\_ptr が、DMI\_atomic(...) から DMI\_function(...) への入力データです。
- in\_ptr が、DMI\_function(...) から DMI\_atomic(...) への出力データ（「返り値」）です。
- DMI\_function(...) の中では、tag の値によってどの read-modify-write を行うかを振り分けることができます。

## ■ 6-3 具体例

### 6-3-1 カウンタ変数をアトミックにインクリメントする

??節で例に挙げた、排他制御変数と条件変数を使ってカウンタ変数 `counter` をインクリメントするプログラムは、ユーザ定義の `read-modify-write` を使うと以下のように記述できます。こちらの方が排他制御変数や条件変数を使うよりはるかに高速です：

### 6-3-2 Allreduce

各 DMI スレッドが保持している `int` 型の整数の総和を求めるプログラムは、ユーザ定義の `read-modify-write` を使って以下のように記述できます：

\*\*\*説明\*\*\*

## ■ 6-4 補足

### 6-4-1 DMI における同期の各種手段の性能比較

さまざまな同期の手段を解説してくる中で、どの手段がどの手段よりも高速であるというようなことを述べてきました。ここでは、ここまでで解説した同期の各手段の性能の関係をまとめておきます。

当然、各手段の性能は、その手段が、どれだけ「プリミティブ」に実装されているかで決定されます。DMI における同期の各手段の関係は Fig.?? のようになっています。Fig.?? からわかるように、DMI の同期において最もプリミティブな同期の手段は、ユーザ定義の `read-modify-write`、`DMI_suspend(...)`、`DMI_wake(...)` の 3 つです。そして、ユーザ定義の `read-modify-write` の特殊な場合として、`compare-and-swap` と `fetch-and-store` が実装されています。さらに、これらの `compare-and-swap` と `fetch-and-store`、`DMI_suspend(...)`、`DMI_wake(...)`、`DMI_read(...)`、`DMI_write(...)` を組み合わせることで排他制御変数が実装されています。一方、バリア操作の API は、ユーザ定義の `read-modify-write`、`DMI_read(...)`、`DMI_write(...)` だけから実装されています。このような DMI における実装上の階層関係から、バリア操作を行うためには排他制御変数と条件変数を組み合わせるよりも、DMI のバリア操作の API を利用の方が高速なことがわかります。また、DMI プログラミングにおいて何らかの同期を実現する場合に、排他制御変数と条件変数を使って同期を実現するよりも、ユーザ定義の `read-modify-write` を使って同期を実現の方が高速なこともわかります。

一般に、分散アプリケーションでは同期は必須であり、同期がボトルネックになることが多々あります。DMI プログラミングで同期を実現する場合、プログラミングの容易さを重視する場合には排他制御変数と条件変数を使えば良いですが、性能を重視する場合にはユーザ定義の `read-modify-write` の機能を利用することが重要です。



# Chap.6 サンプルプログラム

## 1 典型的な DMI プログラミング

- 1-1 DMI プロセスが動的に参加/脱退するプログラム
- 1-2 SPMD 型のプログラム
- 1-3 pthread プログラムから DMI プログラムへの変換

## 2 サンプルプログラムの実行方法



## Chap.7 API 一覽