PARRAY User's Manual (v1.2)

Manycore Software Research Group, School of EECS, Peking University

E-mail: manycore@pku.edu.cn

This manual describes a programming interface called PARRAY (or Parallelizing ARRAYs) that supports system-level succinct programming for heterogeneous parallel systems. PARRAY extends mainstream C and C++ programming with new array typed that contain additional information about the memory type, the layout of the elements in memory and the distribution of data over multiple memory devices. The users only need to learn a unified style of programming for all major parallel architectures including multicore (Pthread/OpenMP), clustering (MPI), distributed memory sharing (G lobal Arrays), GPU (CUDA) and manycore (MIC's OpenMP Offload). The compiler will generate high-performance code according to the typing information contained in the source. This leads to shorter, more portable and maintainable parallel codes, while the programmer still has control over performance-related features necessary for performance optimization.

`Resources of the project can be found at http://code.google.com/p/parray-programming/`

Table of Contents

Listings

1 Introduction

Major vendors have developed low-level programming tool chains that best support their own lines of product: Pthread and OpenMP for multicore , CUDA and OpenCL for GPUs, OpenMP Offload for manycore and MPI for clustering. Combining multiple tools in programming is beyond the grasp of most application developers who are also reluctant to be tied entirely to a specific vender's tool chain.

Figure 1 illustrates a typical scenario of a multicore CPU server connected with two GPU accelerators. A GPU thread regards `dmem` memory as the local device memory, while a CPU thread regards `paged` memory to be the local main memory of the server and `dmem` memory to be the remote device memory of the GPU controlled by that CPU thread. Remote memory typically requires much longer latencies to access, and the bandwidth is often relatively limited.

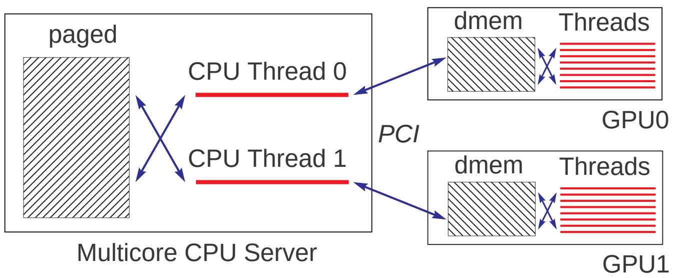

Figure 1 Multicore CPU server with GPU accelerators

PARRAY is implemented as a source-to-source preprocessing compiler. PARRAY may act as a frontend for other runtime libraries. For example, Global Arrays is a PGAS-like tool that supports distributed shared arrays. A programmer may use PARRAY to generate code that invokes Global Arrays. PARRAY extends C language with inserted commands like "`$parray`", "`$copy`", "`$for`" *etc*. These commands correspond to PARRAY pre-defined sub-programs.

PARRAY targets all kinds of multicore, manycore, GPU and cluster parallel threading, all types of memory devices from register file to external storage, and all models of memory-sharing, message-passing and Partitioned-Global-Address-Space communications. The highlights of PARRAY features include:

- **Multi-core threading of CPU (both Pthread and OpenMP)**,
- **Process clustering (MPI)**,
- **PGAS-like communications (Global Arrays)**,
- **GPU including a variety of special types of memory (CUDA)**,
- **MIC manycore threading including the control of vector registers (OpenMP Offload)**.

What distinguishes PARRAY from other programming paradigms is that PARRAY offers a unified style of parallel programming for a variety of hardware devices but still allows the programmer to see their characteristics and control them for hardware-specific performance optimization using succinct code. The design of PARRAY follows the approach of bottom-up abstraction: if a basic operation's algorithm or implementation is not unique (with considerable performance differences), the inclination is to provide more basic operations at a lower level. The intention is to provide a bottom level of abstraction on which performance-minded programmers can directly write very short high-performance code and on which higher-level language features are implementable without unnecessary performance overheads.

1.1 Hello, World

A basic PARRAY program consists of several parts: the header section of usual C and library includes such as "`#include <stdio.h>`" and "`#include <cuda.h>`" as well as specially included basic library "`parray.pa`" with some pre-defined PARRAY sub-programs, invocation to PARRAY's entry C function "`_pa_main`" (which is defined by the PARRAY main-thread context "`$main`", see Section 4.6), and a section of PARRAY code. In the code Listing 1, the main thread, called by the main C function, prints a line of text "`Hello World.`"

Listing 1 Demo Program: hello.pa

```
#include <stdio.h>
$include "parray.pa"
int main(int argc, char *argv[]){
    _pa_main();
    return 0;
}
$main{
    printf("Hello World.\n");
}
```

Implementation: The generated code in "`hello.cpp`" of "`hello.pa`" consists of a considerable number of macros. The C parts of the source program will be preserved by the PARRAY preprocessing compiler and compiled by the underlying C compiler. Such detachment helps make sure PARRAY to be compatible to any C compiler. The generated code for the simplest Hello World program is illustrated in Listing 2.

The first lines are macro definitions generated from command "`$main`" (see Section 4.6). The included user header files are then followed by "`pa.h`". The user main function is placed before the PARRAY code, as the signature of the PARRAY main function "`_pa_main`" is already included in "`pa.h`".

Listing 2 Generated Code: hello.cpp

```
//=== GENERATED OFS CODE ===
#define _pa_dims__PA0_main(_pat) (1)
#define _pa_dstp__PA0_main(_pat) (_pa_dims__PA0_main(_pat)*1)
```

```
......
#include <stdio.h>
......
int main(int argc, char *argv[]){
    _pa_main();
    return 0;
}
//=== GENERATED CODE SIGNATURES ===
void* _pa_pthd_ta__PA0_main(void* _pa_in);
......
```

1.2 Arrays of Parallel Threads

Parallel threads can form an array type. For example, the following thread array type "`M`" defines `n` processes with message-passing communications, type "`P`" defines `n` memory-sharing CPU threads, while "`K`" defines `n` manycore threads on Intel MIC accelerator (default No.0).

```
$parray {mpi[n]} M
$parray {pthd[n]} P
$parray {mic[n]} K
```

PARRAY supports various thread types such as `pthd` for multicore threads on CPU, `cuda` for GPU threads and `mic` for manycore threads on MIC accelerator *etc*. **Note that explicit designation of thread types (as well as memory types) is optional with sub-programming**. The following example code illustrates different types of threads in nested "`$for`" loops.

```
$for i::M {
    $for j::P {
        $for k::K {
            printf("pid=%d, tid=%d, subtid=%d\n",i,j,k);}}}
```

The syntax resembles C's for-loop but instead generates nested arrays of threads. The code first creates `n` message-passing processes, each of which then creates `n` CPU threads, with each creating `n` MIC threads. The MPI's process pid, Pthread's threads tid and MIC's manycore threads subtid index variables are `i`, `j` and `k` respectively.

Implementation: There is no guarantee that an inner-`$for`-loop's code body can read local variables declared in the outer context. This is because the underlying implementation often requires separate C function (*e.g.* for pthread) or explicit list of (*e.g.* for MIC OpenMP Offload) to pass values. PARRAY, however, guarantees to pass all index variables into inner loop bodies. For distributed processes of MPI, this involves sending messages from the invoking thread to the newly created group of processes.

A thread in PARRAY can start an array of new multicore or manycore threads or cluster processes and pass some data through arguments (see Section 5.3). A data array may be located on one process or distributed over multiple processes (see Section 5.5).

Listing 3 Demo Program: hello_parallel.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <mpi.h>
#include <omp.h>
$include "parray.pa"
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    _pa_main();
    MPI_Finalize();
    return 0;
}
#define n 2
$main{
    $for i::mpi[n] {
        printf("Hello from process %d\n", i);
        $for j::pthd[n] {
            printf("\tHello from thread %d,%d\n",i,j);
            $for k::mic[n] {
            printf("\t\tHello from sub-thread %d,%d,%d\n",i,j,k);}}}
}
```

The above code starts from a single main thread that creates 2 processes. Each process creates 2 CPU threads, each of which in turn starts another 2 manycore threads. In total, there will be 8 MIC threads created. Replacing the thread type `mic` with `mpi` will generate 8 sub-processes instead. The result shows a typical run of the code.

Listing 4 Result: mpirun -np 6 hello_parallel

```
Hello from process 0
Hello from process 1
        Hello from thread 0,1
        Hello from thread 1,0
        Hello from thread 1,1
        Hello from thread 0,0
                Hello from sub-thread 1,0,1
                Hello from sub-thread 1,0,0
                Hello from sub-thread 1,1,0
                Hello from sub-thread 1,1,1
                Hello from sub-thread 0,1,0
                Hello from sub-thread 0,1,1
                Hello from sub-thread 0,0,0
                Hello from sub-thread 0,0,1
```

Implementation: PARRAYpre-defines a C macro "_PA_RESERVE_PROCS(n)" to reserve "n" additional idle processes for the following creation of process array. The reserved ones are needed for guaranteeing liveness and allowing the created processes to create more processes without deadlocking the PARRAY scheduler. The macro "_PA_USE_RESERVED_PROCS" indicates that the following created array of processes may use the reserved processes.

1.3 Arrays of Data

The main idea is to extend C array types so that they not only describe the logical dimensional arrangement of array elements but also contain information about how the elements are laid out in the address space of a memory device and how they are distributed over multiple memory devices. For example, if an array of n×n floats is row-major, the index expression of indices i and j is i*n+j. The layout often dictates significant performance differences due to cache-line data locality. If tests suggest to change the layout into column-major, all corresponding index expressions must be modified to j*n+i accordingly. A better approach of PARRAY is to annotate the array's type declaration so that layout changes only affect one line of code.

Multi-dimensional arrays may have more sophisticated layout patterns. For a three-dimensional array of n×n×m floats, row-major index expression for indices i, j and k is i*n*m+j*m+k, while the column-major expression is k*n*n+j*n+i. There may be a mixture of row-major layout for the rightmost dimension and column-major layout for the middle dimension where the index expression becomes i*m+j*n*m+k. To handle such sophistication of different layouts, PARRAY generalizes traditional array types with location information. The following array type "A" in paged main memory managed by OS has three dimensions

```
$parray {paged float[[n][n]][m]} A
```

and consists of n*n*m elements. The number of elements is denoted as a PARRAY expression "$size(A)". It also extends C's array type with additional information about paged memory type where its array objects will be allocated. The following commands declare two type-A array objects x and y as pointers and allocate memory to the pointers using the corresponding library calls of the memory type.

```
float *x,*y;  $malloc A(x,y)
```

Note that the commands are the same as "`$create A(x,y)`" in shorthand.

Implementation: Memory of type `paged` uses standard C functions "`malloc`" and "`free`" to allocate and release. PARRAY supports various other memory types including `dmem` for GPU device memory, `micmem` for MIC memory *etc.*

Unlike C language, type "`A`" nests its first two dimensions together, and is also regarded as a two-dimensional type. The size "`$size(A_0)`" of the column dimension "`A_0`" is `n*n`, which is split into two sub-dimensions "`A_0_0`" and "`A_0_1`" of size `n`.

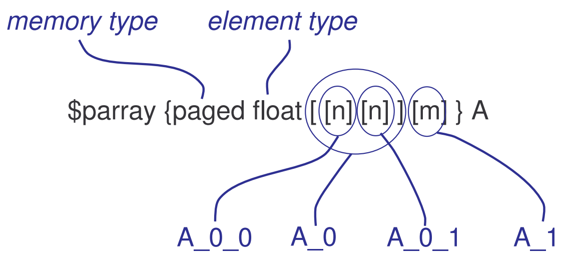

Figure 2 Array type and naming of dimensions

The offset expression of a dimension "`A_0_0`" is denoted as "`$A_0_0[k]`" for some index `k`. Two-dimensional indexing is allowed for 2D types such as "`$A[i][j]`" and "`$A_0[i][j]`". Table 1 identifies the PARRAY expressions derived from "`A`". Figure 3(a) shows the memory layout of "`A`".

T=	A	A_0	A_1	A_0_0	A_0_1
$size(T)	n*n*m	n*n	m	n	n
$T[k]	k	k*m	k	k*n*m	k*m
$T[i][j]	i*m + j	i*n*m + j*m			
$T[[i][j]][k]	i*n*m + j*m + k				

Table 1 PARRAY size and offset expressions derived from "`A`" for `i`, `j` and `k` bounded by the sizes of their corresponding dimensions.

Type "`A`" is row-major, and the offsets of its row dimension "`A_1`" are contiguously laid out in memory. An array type may take other shapes. PARRAY allows array dimensions to refer to existing types. This simple mechanism has powerful expressiveness. The following type "`B`" also consists of `n*n*m` elements:

```
$parray {dmem float[[#A_0_1][#A_0_0]][#A_1]} B
```

but is allocated in GPU's device memory. Its row dimension "`B_1`" has the same offsets as "`A_1`" (according to dimensional reference "`#A_1`"), but the sub-dimensions of the column dimension are swapped. For example, the index expression "`$B_0[k]`" is calculated by first decomposing the index `k` (where `k<n*n`) into the quotient `k/n`

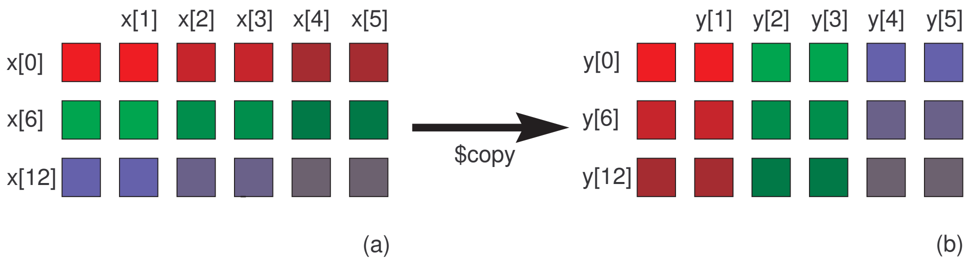

Figure 3 Memory layout and "`$copy A(x) to B(y)`" where `n=3` and `m=2`.

of division and the remainder `k%n` of modulo, and then applies the offsets of "`A`" on the two parts (see Table 2 for other index expressions).

```
$B_0[k] = $B_0_0[k/n] + $B_0_1[k%n]
        = $A_0_1[k/n] + $A_0_0[k%n]
        = k/n*m + k%n*n*m
```

T=	B	B_0	B_1	B_0_0	B_0_1
`$size(T)`	`n*n*m`	`n*n`	`m`	`n`	`n`
`$T[k]`	`k/m/n*m + k/m%n*n*m + k%m`	`k/n*m + k%n*n*m`	`k`	`k*m`	`k*n*m`
`$T[i][j]`	`i/n*m + i%n*n*m + j`	`i*m + j*n*m`			
`$T[[i][j]][k]`	`i*m + j*n*m + k`				

Table 2 PARRAY size and offset expressions derived from "`B`" for `i`, `j` and `k` bounded by the sizes of their corresponding dimensions.

The following PARRAY command "`$copy`" duplicates `n*n*m` floats at address `x` in the main memory to address `y` in GPU device memory so that the array element "`y[$B[k]]`" becomes a copy of "`x[$A[k]]`":

```
$copy A(x) to B(y).
```

If we consider every two adjacent elements as a unit, the layout of `y` is exactly a matrix transposition of `x` (see Figure 3).

We include another simple code to illustrate the basic notions of PARRAY. Code Listing 5 first declares a two-dimensional array type "`A`" of integers in the normal OS-managed "`paged`" memory. The size of each dimension is denoted by a variable "`n`" whose value is determined in runtime. Following C's row-major convention, the right (row) dimension of "`A`" is contiguous in memory, while the left (column) dimension has a regular gap of length "`n`".

The definition of another array type "`B`" looks different though. It indicates that the column dimension of "`B`" refers to the row dimension "`A_1`" of "`A`", while the row dimension of "`B`" refers to the column dimension "`A_0`", essentially transposing the original array. Similar to "A", elements of "B" are also integers in "paged" memory. The exact naming convention of the dimensions is explained in Section 2.3.

Listing 5 Demo Program: hello_arrays.pa

```
#include <stdio.h>
#include <stdlib.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {paged int[[n][n]][m]} A
$parray {[[#A_0_1][#A_0_0]][#A_1]} B
$main{
    int n=3, m=2;
    $create A(a,b)
    $for i::A(a) { *a=i; }
    $copy A(a) to B(b)
    printf("Hello Array World:\n");
    $for i::A(a,b) {
        printf("i=%d, a[i]=%d, b[i]=%d\n", i, *a, *b);}
    $destroy A(a,b)
}
```

The array objects "a" and "b" are declared as integer pointers, and each is allocated with n*n integers in OS-managed page memory. The integer array elements of "a" are initialized to their offset values and copied to "b" using the PARRAY command "$copy", but the order is twisted with rows and columns transposed. The array objects are freed by the PARRAY command "$destroy".

Listing 6 Result: hello_arrays

```
Hello Array World:
i=0, a[i]=0, b[i]=0
i=1, a[i]=1, b[i]=1
i=2, a[i]=2, b[i]=6
i=3, a[i]=3, b[i]=7
i=4, a[i]=4, b[i]=12
i=5, a[i]=5, b[i]=13
i=6, a[i]=6, b[i]=2
i=7, a[i]=7, b[i]=3
i=8, a[i]=8, b[i]=8
i=9, a[i]=9, b[i]=9
i=10, a[i]=10, b[i]=14
i=11, a[i]=11, b[i]=15
```

```
i=12, a[i]=12, b[i]=4
i=13, a[i]=13, b[i]=5
i=14, a[i]=14, b[i]=10
i=15, a[i]=15, b[i]=11
i=16, a[i]=16, b[i]=16
i=17, a[i]=17, b[i]=17
```

1.4 Arrays of Communications

Section 1.3 describes array layout in a single memory device. The idea is further elaborated to represent data distribution. For a three-dimensional array of n*n*m, if we indicate that the middle dimension is distributed over n processes, then the indices i, j and k will define two index expressions for any element: a simple expression j indicating the pid of the process on which that element is located, and the expression i*m+k indicating the element's offset in the address space of that process. With the location of every element known, it is then easy for the compiler to generate code of memory allocation, deallocation, and communication using whichever vender libraries.

Dimensional reference can be used to represent data distribution over multiple memory devices. This is the case when a type contains a mixture of thread dimensions and data dimensions. The following two array types also have n*n*m floats, but they are distributed over n message-passing communicating processes:

```
$parray {[[#M][#A_0_1]][#A_1]} C
$parray {[[#A_0_1][#M]][#A_1]} D
```

where type "C"'s column sub-dimension "C_0_0" is distributed, while type "D"'s row sub-dimension "D_0_1" is distributed. Matrix transposition (in the unit of every two floats) over message-passing processes is better known

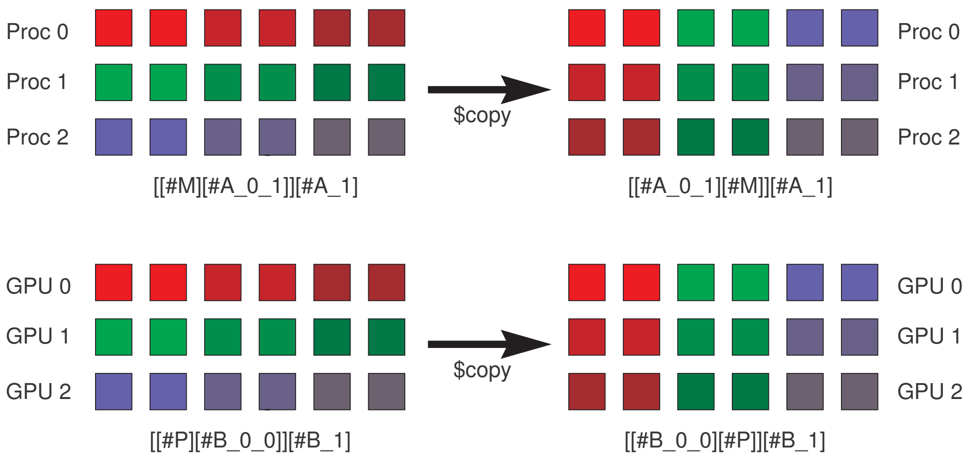

Figure 4 All-to-all communications for n=3 and m=2

as all-to-all communication and corresponds to an MPI library call "`MPI_Alltoall`". Such an operation must be performed by `n` processes collectively in code like:

```
$for M {
    ......
    $copy C(x) to D(y)
    ......
}
```

Data may also be distributed over several GPU devices connected to a server. The data communication with each device is performed by a CPU thread controlling the GPU. The following types describe arrays distributed over `n` GPU devices. The all-to-all between "`E`" and "`F`" is performed among GPU devices, and the generated code uses CUDA instead of the MPI library.

```
$parray {[[#P][#B_0_0]][#B_1]} E
$parray {[[#B_0_0][#P]][#B_1]} F
```

The "`$copy`" operation is performed by `n` CPU threads collectively, after each initialising a GPU whose device memory can then be accessed:

```
$for k::P {
    ......
    INIT_GPU(k)
    ......
    $copy E(x) to F(y)
    ......
}
```

2 Basic Array Types

PARRAY extends C and C++ with new array types. These types not only describe the number of dimensions, the dimension sizes and the type of each element, but also convey information about data layout and distribution to the compiler, which can then be used for generating efficient code.

2.1 Memory Types (`paged/pinned/mpimem/dmem/smem/rmem/micmem/vmem/mmap/gamem`)

PARRAY currently supports a number of pre-defined memory types. The code generation for allocation and release of such an array object requires lower-level libraries as well as their corresponding header files. Note that `pinned` and `mpimem` memory types are inter-operable on systems installed with GPU-Direct.

Memory Type	Descrption	Allocation	Release
`paged`	Linux paged virtual memory (`stdlib.h`)	`malloc`	`free`
`pinned`	CUDA page-lock main memory (`cuda.h`)	`cudaHostMalloc`	`cudaFreeHost`
`mpimem`	MPI page-lock memory for Infiniband (`mpi.h`)	`MPI_Alloc_mem`	`MPI_Free_mem`
`dmem`	Device memory of CUDA (`cuda.h`)	`cudaMalloc`	`cudaFree`
`smem`	Shared memory of a CUDA block (`cuda.h`)	`__shared__`	*automatic*
`rmem`	Register file of a CUDA thread (`cuda.h`)	*direct declaration*	*automatic*
`micmem`	Memory of MIC (`omp.h`)	`#pragma for`	`#pragma for`
`vmem`	Vector register file of MIC (`immintrin.h`)	`__m512d`	*automatic*
`mmap`	Linux virtual memory for file image (`sys/mman.h`)	`mmap`	`munmap`
`gamem`	Global Arrays PGAS memory (`ga.h`)	`NGA_Create`	`GA_Destroy`

2.2 Thread Types (`mpi/pthd/omp/cuda/mic`)

PARRAY currently supports a number of pre-defined thread types. To use these types of threads, the programming environment must have the necessary libraries installed and necessary header files included.

Thread Type	Descrption
`mpi`	MPI processes for clustering (`mpi.h`)
`pthd`	Pthread threads for CPU or MIC mutli-threading (`pthread.h`)
`omp`	OpenMP threads for CPU or MIC many-threading (`omp.h`)
`cuda`	CUDA threads for GPU (`cuda.h`)
`mic`	OpenMP parallel threads for MIC (`omp.h`)

2.3 Declaring Array Types (`parray`)

Array types in PARRAY are different from those in C. The first obvious distinction is the separation between array objects and their types. An array type in PARRAY is declared with a directive command "`$parray`". An actual array object is usually declared as a C pointer to which memory space can be allocated by a directive command "`$create`" (see Section 3) that takes its array type as a parameter. That means a C pointer and its associated data may be interpreted to different array types in different parts of the same code.

Passing an array as an argument to a function or accessing its elements must involve both the object (*i.e.* the pointer to the starting address) and its array type by which the array object is interpreted. An array type can be used for type reference or generating index expressions only and does not create any real array objects.

Simple type declaration of arrays has the following command (see Section 5 for declaration of more advanced array types):

`$parray` {*mtt element_type dimension_tree*} *type_name*

where "*mtt*" denotes a memory/thread type, "*element_type*" is the C type (default as being empty) of each element, "*dimension_tree*" is a dimension tree (see Section 2.3), and finally "*type_name*" is the name of the type.

The "`$parray`" command declares a named array type. It is also possible to declare on-spot array types without names. This is useful when an array type is used only once and need not be referred by other array types. Anonymous array types use the following command:

`$`*mtt element_type dimension_tree*.

Implementation: Type declaration "`$parray`" with a type name can be placed anywhere in a program text. The generated C macros of all type declaration are collected at the beginning of the generated code. As an anonymous array type has no user-defined name, a temporary name is assigned at the program locations of their declaration by the compiler. The declaration itself becomes the identity of the type.

Type declaration usually requires "`$`" to be distinguished from C program text, but for a PARRAY command (*e.g.* "`$for`") that expects an array type, "`$`"is omissible. Code Listing 28 contains an anonymous thread array type in a "`$for`" command.

Example: Let us first consider a simple type definition with 3x3x2 floats:

```
$parray {paged float[[3][3]][2]} A
```

where the memory type "`paged`" indicates that the arrays of this type are in the main memory under OS paging management, "`float`" is the element type, and the offsets observe the row-major convention. In practice, dimension sizes can be any integer expressions including variables whose values are determined in runtime.

The dimensions of PARRAY types are organized in a tree structures, conceptually reflecting the memory hierarchy of heterogeneous parallel systems. Unlike the hierarchical dimension structures in other array notations, definition here can represent the hierarchical data organization within the same memory device as well as their

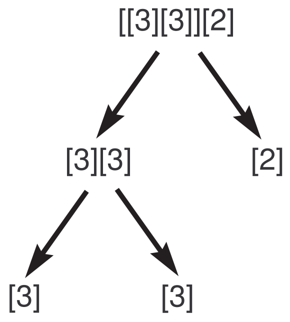

Figure 5 Dimension tree of [[3][3]][2]

distribution over multiple devices. Thus the three-dimensional array type "`A`" is also two-dimensional with 9x2 floats or one-dimensional with 18 floats (see Figure 5).

Example: A dimension name "`foo_0_9`" contains a root type name "`foo`". The first suffix "`_0`" indicates that the dimension belongs to the 0-th (or leftmost) sub-dimension of "`foo`". The total number of sub-dimensions is restricted to 10 for each dimension. A suffix "`_99`" is therefore not allowed. Thus the dimension "`foo_0_9`" indicates it is the last sub-dimension of "`foo_0`". Likewise, at the uppermost level, the dimension "`foo_1`" is the second dimension of that level. The declaration of array type "`foo`" might look like the following where "`foo_0_9`" denotes the dimension with size 19.

```
$parray {pinned float[[10][11][12][13][14][15][16][17][18][19]] [20]} foo
```

Implementation: As PARRAY syntax is in close combination with C or C++ syntax, code generation must be conducted under a number of syntactical restrictions of the host language. The programmer is advised not to use prefix "`_pa`" in variable and function names. Dimension names are context-sensitive. Declaring an array type within the scope of a sub-program will not conflict with any array type outside of the sub-program even if they share the same dimension name. Certain context-sensitive prefix is inserted by the compiler in the generated code.

If an array type is expected (for example in a "`$for`" command), a context-sensitive prefix will be automatically added. If this is not the case but the code still needs to pass a type name around, the context-sensitive prefix can be inserted with an expression "`$`*type_name*" by the programmer.

2.4 Offset Expression of Array Elements

A PARRAY type can be deemed as a general C macro that gives rise to expressions that compute the offsets of indices. The command for such an expression is as follows:

`$`*type_name dimension_tree*

where "*dimension_tree*" denotes the dimension tree of nested indices. For example, according to a simplified representation, "`$A[[i][j]][k]`" corresponds to "`i*6+j*2+k`", "`$A[i][j]`" corresponds to "`(i*2+j)`", while "`$A[i]`" corresponds to the index variable "`i`" itself. Let "`A_0`" denote the column dimension "`A`" and "`A_1`" denote the row dimension. Then the expression "`$A_0[i][j]`" corresponds to "`(i*6+j*2)`", while "`$A_0_1[i]`" corresponds to "`(i*2)`". Note that it is the programmer's responsibility to ensure that every index is an in-scope integer that is non-negative and less than its dimension size. The value of an index expression with out-of-scope indices is unspecified.

2.5 Size of Dimension (`size`)

The PARRAY command "`$size`" returns the number of threads in the current thread-array context. Such a context is defined by the immediate "`$for`" or other context-defining environment such as "`$main`". If the current context is "`$main`", then the size is 1. The global context "`$global`", however, has no dimension or dimension size. In general, the dimension-size expression uses the command "`$size(`*type_name*`)`". It returns the size of that specific dimension.

2.6 Element Type (`elm`)

The PARRAY command "`$elm(`*type_name*`)`" returns the C element type of the array type "`type_name`". Unspecified memory or thread type returns "`void`".

Code Listing 25 shows a sub-program that uses such an expression, which becomes useful when the element type of an array type in a sub-program's arguments depends on the context of insertion and not statically available to the sub-program.

2.7 Memory and Thread Type (`mtt`)

The PARRAY command "`$mtt(`*type_name*`)`" returns the MTT of the array type "*type_name*". Unspecified element type returns "`any`".

2.8 Number of Sub-Dimensions (`ndim`)

The PARRAY command "`$ndim(`*type_name*`)`" returns the number of sub-dimensions or 0 if no sub-dimension exists. For example, type "`A`" in Section 2.3 has two dimensions, and therefore "`$ndim(A)`" is syntactically replaced by the number " 2 " in compilation.

2.9 Dimensional Type Reference (#)

One of the key features of PARRAY is that a dimension can refer to another type's dimension and follow that dimension's offset indexing. The command for dimensional type reference is as follows.

$$\textit{dimension}\ \#\ \textit{referred\_type}_1\ \#\ \cdots\ \#\ \textit{referred\_type}_n$$

where *dimension* is a C expression (not necessarily a constant) or other forms of dimensions (such as displacement), and each *referred_type$_i$*'s offset function transforms the left-hand side's offset. Multiple referred types lead to functional composition.

Example: For an existing type "`$parray {pinned float[[3][3]][2]} A`", valid type declarations with references may come in different forms. A one-dimensional array type "`B`" has a confined range of indices 0~6:

```
$parray {dmem float[7 # A_0]} B.
```

The offsets of the above array type satisfy a condition "`$B[i] == $A_0[i % 7]`" where "`$A_0[i] == i % 9 * 2`". As "`B`" is one-dimensional, the overall offset "`B[i]`" and its only dimension's offset "`B_0[i]`" are equal. A related but different type is declared as follows:

```
$parray {dmem float 7 # A_0} C.
```

This type satisfies the condition "`$C[i] == $A_0[i % 7]`". However, the sub-dimension "`C_0`" does not exist. If the dimension size is an expression (instead of an integer), brackets should be added to mark it from the element type (*e.g.* "`$parray {dmem unsigned int (N) # A_0} C`" indicating that "`(N)`", as a dimension size, is not part of the element type). A dimensional type reference may have sub-dimensions, which may in turn have their references. The following type decomposes "`A_0`" into two sub-dimensions:

```
$parray {dmem float[2][2] # A_0} D.
```

The above type satisfies the condition "`$D_0[i] == $A_0[i % 2 * 2]`". Type reference is often used to re-arrange the order of the dimensions. The following type declaration swaps the two column sub-dimensions of "`A`":

```
$parray {paged float[#A_0_1][#A_0_0]} E.
```

The above type satisfies "`$E_0[i] == $A_0_1[i]`". A pure type reference like the column dimension "`#A_0_1`" without a designated dimension size also cites the entire sub-dimension tree of "`A_0_1`". This is more evident in the following type declaration:

```
$parray {paged float # A_0} F.
```

The above type satisfies the condition "`$F_0[i] == $A_0_0[i]`". Despite the fact that the type does not directly have any dimension, "`F_0`" and "`F_1`" are inherited from "`A_0_0`" and "`A_0_1`" respectively. Note that multiple type references can be composed in a sequence, but only the first (*i.e.* leftmost) reference determines the dimension size. Code Listing 7 illustrates the above examples.

Listing 7 Demo Program: type_references.pa

```
#include <stdio.h>
#include <stdio.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {pinned float[[3][3]][2]} A
$parray {dmem float[7#A_0]} B
$parray {dmem float 7#A_0} C
$parray {dmem float [2][2]#A_0} D
$parray {paged float [#A_0_1][#A_0_0]} E
$parray {paged float #A_0} F
$main{
    for (int i=0;i<$size(C);i++) printf ("B_0[%d]=%d ",i,$B_0[i]);
    printf("\n");
    for (int i=0;i<$size(C);i++) printf ("C[%d]=%d ",i,$C[i]);
    printf("\n");
    for (int i=0;i<$size(D_0);i++) printf ("D_0[%d]=%d ",i,$D_0[i]);
    printf("\n");
    for (int i=0;i<$size(E_0);i++) printf ("E_0[%d]=%d ",i,$E_0[i]);
    printf("\n");
    for (int i=0;i<$size(F_0);i++) printf ("F_0[%d]=%d ",i,$F_0[i]);
    printf("\n");
}
```

Listing 8 Result: type_references

```
B_0[0]=0 B_0[1]=2 B_0[2]=4 B_0[3]=6 B_0[4]=8 B_0[5]=10 B_0[6]=12
C[0]=0 C[1]=2 C[2]=4 C[3]=6 C[4]=8 C[5]=10 C[6]=12
D_0[0]=0 D_0[1]=4
E_0[0]=0 E_0[1]=2 E_0[2]=4
F_0[0]=0 F_0[1]=6 F_0[2]=12
```

2.10 Dimensional Offset Displacement (`disp`)

Usually a dimension's offset starts from 0, but this can be altered with additional displacement. A dimensional offset displacement has the following command:

`disp`(*offset_displacement*)

where "*offset_displacement*" is a C expression (not necessarily a constant) that is added to the dimension's offset. A displacement itself has dimension size 1 but allows any index.

Example: A typical usage of this notation is to depict a sub-space window from a regular multi-dimensional space. Consider a two-dimensional space

`$parray {paged float[100][100]} DATA`.

A 80x80 window can be declared in the middle of the space:

`$parray {paged float[80#disp(10)#DATA_0][80#disp(10)#DATA_1]} WINDOW`.

Either the column or row dimension now has dimension size 80. In terms of offset, either index (to be transformed by the original dimensional offset functions of "`DATA_0`" and"`DATA_1`") is shifted with a displacement of 10. Figure 6 illustrates the relation between "`WINDOW`" and the original "`DATA`" where the offsets satisfy "`$WINDOW[i][j]=$DATA[i+10][j+10]`" for `i<80` and `j<80`.

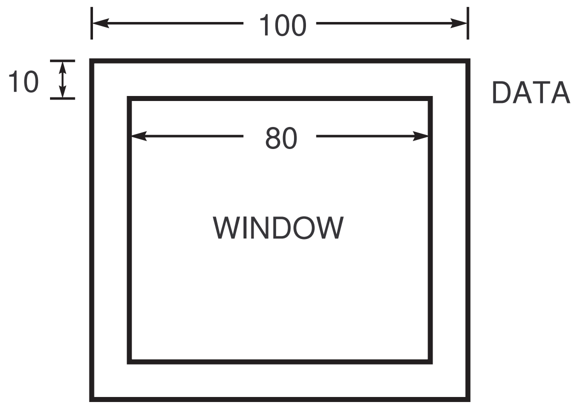

Figure 6 Displaced 2D window inside a regular 2D matrix

Code Listing 9 copies a 80x80 window within a 100x100 square to a regular 80x80 array. The data transfer is performed for 80 rows, each with 80 elements in contiguous layout.

Listing 9 Demo Program: displacement.pa

```
#include <stdio.h>
#include <stdlib.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {paged float[100][100]} DATA
$parray {paged float[80#disp(10)#DATA_0][80#disp(10)#DATA_1]} WINDOW
$parray {paged float[80][80]} COPY
```

```
$main{
    $create DATA(data), COPY(copy)
    $copy WINDOW(data) to COPY(copy)
    printf("WINDOW[0][0]=DATA[10][10]=%d.\n", $WINDOW[0][0]);
    printf("stp(DATA)=%d, stp(DATA_0)=%d, stp(DATA_1)=%d.\n",
        $stp(DATA),  $stp(DATA_0), $stp(DATA_1));
    printf("stp(WINDOW)=%d, stp(WINDOW_0)=%d, stp(WINDOW_1)=%d.\n",
        $stp(WINDOW),  $stp(WINDOW_0), $stp(WINDOW_1));
    $destroy DATA(data), COPY(copy)
}
```

Listing 10 Result: displacement

```
WINDOW[0][0]=DATA[10][10]=1010.
stp(DATA)=1, stp(DATA_0)=100, stp(DATA_1)=1.
stp(WINDOW)=0, stp(WINDOW_0)=100, stp(WINDOW_1)=1.
```

Example: Sometimes offset displacements form a cycle in a range. Code Listing 11 shows how to write array type that has offsets shifted 3 elements to the right cyclically (see Figure 7).

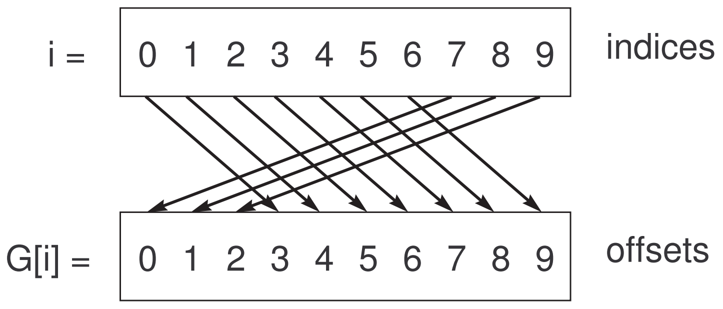

Figure 7 Cyclic Displacement of Offsets

Listing 11 Demo Program: cyclic.pa

```
#include <stdio.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {[10 # disp(3) # dim(10)]} G
$main{
    for (int i=0; i<10; i++) printf("G[%d]=%d ", i, $G[i]);
    printf("\n");
}
```

Listing 12 Result: cyclic.pa

```
G[0]=3 G[1]=4 G[2]=5 G[3]=6 G[4]=7 G[5]=8 G[6]=9 G[7]=0 G[8]=1 G[9]=2
```

2.11 User-Defined Offset Functions (func)

In some applications with irregular data forms, conventional notations are not sufficient in defining the offsets. The programmer may directly write their own offset functions as macros. The array-type notation has the following command:

func(arg_1, $\cdots$, arg_n) *macro_def*

where arg_i is the macro's arguments, and *macro_def* is the macro's body.

Example: Code Listing 13 illustrates an array type whose offsets depend on the value of an indexing array "offset". The code stores large sparse arrays in a compressed form. Only offsets of the accessed elements are recorded in an array "offset" whose values (as the offsets of another array type) can be determined in runtime. Note that "printf" usually computes arguments from the right to the left.

Listing 13 Demo Program: func.pa

```
#include <stdio.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
int id=0;
int offset[100];
int access(int i) {
    for (int j=0; j<id; j++) if (offset[j]==i) return j;
    offset[id]=i;
    return id++;
}
$parray {[1000# func(i)access(i)]} DATA
$main{
    printf("DATA[4]=%d, DATA[5]=%d, DATA[6]=%d\n",
        $DATA[4], $DATA[5], $DATA[6]);
}
```

Listing 14 Result: func.pa

```
DATA[4]=2, DATA[5]=1, DATA[6]=0
```

The the indexing function “access” can be further improved to reflect more sophisticated searching mechanisms. When both conventional PARRAY notations and “func” are applicable, the former are preferred for better readability.

3 Basic Memory Allocation and Data Movement

Both performance and energy consumption are highly correlated to the locality of data and their movement across different memory devices. These factors are often the most critical factor for optimization in many computing tasks. In PARRAY, the programmer has been given a substantial amount of flexibility to control such low-level features. This section will describe the basic forms of memory allocation and data movement.

3.1 Memory Allocation of Arrays (`declare/create/destroy/malloc`)

The declaration, allocation and release of data arrays use the following commands:

```
$declare type_name(pointer_address)
$malloc type_name(pointer_address)
$create type_name(pointer_address)
$destroy type_name(pointer_address)
```

where *type_name* is a type or a specific dimension, and *pointer_address* is the pointer variable to be allocated. Command "`$declare`" declares an array of size `$size(`*type_name* in the current static environment. If an array is declared in the body of a loop, only one array object will be allocated during looping and will be released automatically at exit. Command "`$malloc`" allocates contiguous memory in runtime and assigns the starting address to a previously declared pointer variable *pointer_address*, see Section 7.2. The command allocates an array object whenever it is executed. For example, a pointer variable may be declared globally and allocated in the main thread but accessed by other sub-threads in a different lexical context. Command "`$create`" allocates a contiguous address space starting from *pointer_address*. If both static and dynamic allocation are possible in the current static environment, "`$create`" prefers static allocation. Command "`$destroy`" releases the space of an array. If memory allocation is unsuccessful, the NULL pointer is assigned to *pointer_address*.

3.2 Data Copying (`copy`)

PARRAY types provide the preprocessing compiler with enough information to generate efficient code that can copy one array in a certain distribution and layout to another array in a different distribution and layout. Copying one array's data to another array uses the following command:

```
$copy type_source(ptr_source) to type_target(ptr_target)
```

where *ptr_target* denotes the name of the target array (normally a pointer), *type_target* denotes the type name of the target array, *ptr_source* denotes the name of the source array (normally a pointer), and *type_source* denotes the type name of the source array.

The command copies every element "*ptr_source*`[$`*type_source*`[i]]`" in the source array to the element "*ptr_target*`[$`*type_target*`[i]]`" in the target array.

Implementation: The implementation is to call various different patterns of data transfer by checking the dimension structures and certain features of the array types. The principle of optimization is to reduce overheads and maximize communication granularity. The exact mechanism of implementation for data-transfer patterns is version-related and subject to future changes. This basic operation is implemented as a sub-program in the library "`parray.pa`".

Code Listing 15 tests the bandwidth between host pinned (*i.e.* page-lock) memory and GPU device memory. If the array offsets are not contiguous, more-than-one memory copying command will be needed. In this particular case, as the column dimension of the dmem array is not contiguous but the row dimension is, the size of each contiguous unit coincides with the length of each row. The PCI bandwidth (across PCI) heavily depends on communication granularity and will be considerably affected if the contiguous unit is smaller than 2MB.

Listing 15 Demo Program: data_transfer.pa

```
#include <stdio.h>
#include <cuda.h>
$include "parray.pa"
#define SIZE 4096
int main(int argc,char *argv[]){_pa_main(); return 0;}
$parray {paged double[[SIZE/16][16]][SIZE]} HOST
$parray {paged double[[#HOST_0_1][#HOST_0_0]][SIZE]} DEV
$main{
    _PA_INIT_GPU(0);    // setting GPU device 0
    $create HOST(host), DEV(dev)
    _PA_CREATE_TIMER
    $copy HOST(host) to HOST(dev)
    _PA_TIME_NOW
    float sec = (float)_PA_TIME_DIFF(0,1);
    float gb = (float)$size(HOST)*sizeof($elm(HOST))/1024/1024/1024;
    printf("%.3f sec %.3fGB  %.3f GB/s\n", sec, gb, gb/sec);
    $destroy HOST(host), DEV(dev)
}
```

3.3 Arrays of File Images (`mmap`)

File image allocated by Linux command "`mmap`" maps files or devices into memory for input and output. PARRAY uses the keyword "`mmap`" to denote this kind of virtual memory. From coding point of view, it is regarded as *local* memory since once it is created, accessing such memory is just like accessing local memory.

An "`mmap`" array type depends on several preprocessed variables (see Section 6.5) which should be set before

"`$create`" or "`$malloc`":

Variable	Descrption	Default Value
`mmap_base`	starting address of file image	`NULL` (*determined by system*)
`mmap_prot`	read/write protection	`PROT_WRITE` (*write protection*)
`mmap_flag`	type of file image	`MAP_SHARED` (*shared by all processes*)
`mmap_file`	file handler	`NULL` (*no assignment*)
`mmap_offset`	offset of file image	`0` (*no offset*)

Implementation: File image allocated by Linux command "`mmap`" on demand-paging in virtual memory, as the file contents are initially on the disks and only loaded when the very pages are accessed by the process. The Linux "`munmap`" (*i.e.* "`$destroy`" in PARRAY) is required to write back the image after the accesses are complete. To use this memory type, necessary header files must be included.

Code Listing 16 illustrates a simple program that swaps the first two bytes of a file.

Listing 16 Demo Program: mmap.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
$include "parray.pa"
int main(int argc,char *argv[]){_pa_main(); return 0;}
$parray {mmap char[2]} HOST
$main{
   int fd = open("./mmap.data", O_RDWR);
   if (fd<0) {printf("file open error\n"); exit(1);}
   $var mmap_file(fd)
   $create HOST(host)
   char tmp; tmp=host[0]; host[0]=host[1]; host[1]=tmp;
   printf ("./mmap.data: %c %c\n", host[0], host[1]);
   $destroy HOST(host)
   close(fd);
}
```

3.4 PGAS Global Arrays (`gamem`)

Partitioned Global Address Space (or PGAS) is a popular style of parallel programming and a tradeoff between memory sharing in which data locality is transparent to the programmer and message passing in which communications are mainly two sided. In PGAS programming, the source code may still interfere with the affiliation of data with their processes and at the same time enjoy one-sided communications completely free of any commands issued by the passive participants during communication. Message-passing libraries such as MPI today do not fully support this kind of clean one-sidedness. Global Arrays is a PGAS-like library designed to work with MPI.

PARRAY uses the keyword "`gamem`" to denote memory allocated by Global Arrays. Although "`gamem`" memory is physically distributed, it is regarded as *local* memory by PARRAY since the program code can access all the addresses as if they exist locally. In contrast, Section 5.5 will explain how truly distributed memory (both logically and physically) is handled in PARRAY.

Code Listing 17 illustrates the creation of a Global Arrays memory on a group of 4 MPI processes (see Section 4 for command "`$for`") and a local "`paged`" memory on process 0. Header file "`ga.h`" and necessary initialization as well as finalization are required for Global Arrays. The code sets the initial values of the "`paged`" array "`buf`" and copies them to and from the Global Arrays memory "`data`".

Implementation: Unlike arrays of other memory types, the handle of a Global Arrays memory is an integer instead of a pointer. Accesses to its elements must be performed with the "`$copy`" command instead of pointer arithmetic. Currently only array types without type references and restricted data-transfer patterns are supported. The implementation is contained in library "`parray.pa`".

Listing 17 Demo Program: gamem.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include "ga.h"
$include "parray.pa"
int main(int argc,char *argv[]){
   MPI_Init(&argc, &argv);
   GA_Initialize();
   _pa_main();
   GA_Terminate();
   MPI_Finalize();
   return 0;
}
$parray{gamem int[3][4][2]} DATA
$parray{paged int[24]} BUF
```

```
$main{
    $for k::mpi[4] {
        $create DATA(data)
        GA_Print_distribution(data);
        if(k==0) {
            $create BUF(mybuf)
            $for i::BUF(mybuf) { *mybuf=i; }
            $copy BUF(mybuf) to DATA(data)
            $for i::BUF(mybuf) { *mybuf=-1; }
            $copy DATA(data) to BUF(mybuf)
            $for i::BUF(mybuf) { printf("%d ",*mybuf); }}
    }
}
```

This code must be executed on no-less-than 5 processes (including the main process). If given more processes, PARRAY will automatically choose 5 of them.

Listing 18 Result: mpirun -np 6 gamem

```
Array Handle=-1000 Name:'' Data Type:float
Array Dimensions:3x4x2
Process=0         owns array section: [0:2,0:1,0:0]
Process=1         owns array section: [0:2,0:1,1:1]
Process=2         owns array section: [0:2,2:3,0:0]
Process=3         owns array section: [0:2,2:3,1:1]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

3.5 Mapping of Data Arrays (for)

Applying (or mapping) an operation to all elements of data arrays is handled by the "`$for`" command, which is similar in functionality as the "map" command in other languages. Here we assume that the programmer does not specify explicitly the kind of processor and the number of threads to be used. Instead the threads will be assigned automatically according to the memory types.

The basic syntax of the "`for`" command for mapping is as follows:

$$\texttt{\$for}\ \ type\_name(data\_arg_1, data\_arg_2, \cdots, data\_arg_n)\ \ \{\ \ code\ \ \}$$

where the data arguments are the pointers to the arrays of data type "*type_name*". The element type of each pointer $data\_arg_i$ must be the same as the element type of *type_name*. In generated code, each pointer $data\_arg_i$ will be moved to ($data\_arg_i$ + \$*type_name*[`$tid`]) for each index "`$tid`" (see Section 4.8) of element as "$*\,data\_arg_i$" (or "$data\_arg_i$`[0]`").

Implementation: If the header file "`omp.h`" is present, PARRAY may try to parallelize a loop in OpenMP style. Whether to parallelize a loop depends on the actual implementation.

Code Listing 19 illustrates the mapping of an array in the main memory.

Listing 19 Demo Program: for_data.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {paged int[4]} A
$main{
    $create A(a)
    $for k::A(a) { *a=k+1; }
    $for k::A(a) { printf("a[%d]=%d\n", k, *a); }
    $destroy A(a)
}
```

The result may show the order of mapping non-deterministically.

Listing 20 Result: for_data

```
a[0]=1
a[2]=3
a[1]=2
a[3]=4
```

3.6 Mapping of Data Arrays on GPU (`for`)

Code Listing 21 illustrates the mapping of an array in GPU's device memory. PARRAY will choose to run the code body of the "`$for`" command in a CUDA kernel. The macro "`_PA_INIT_GPU`" (defined in a header file "`include/pa_cuda.h`") initializes the GPU of device 0.

Listing 21 Demo Program: for_dmem.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
```

```
$parray {paged int[4]} A
$parray {dmem #A} B
$main{
    _PA_INIT_GPU(0);
    $create A(a), B(b)
    $for A(a) { *a=0; }
    $copy A(a) to B(b)
    $for k::B(b) { *b=k+1; }
    $copy B(b) to A(a)
    $for k::A(a) { printf("a[%d]=%d\n",k,*a); }
    $destroy A(a), B(b)
}
```

3.7 Mapping of Data Arrays on MIC (for)

Code Listing 22 illustrates the mapping of an array in Intel MIC's on-card memory. The pointer "a" in the code has dual types for both the main memory (of memory type "paged") and the MIC memory (of memory type "micmem"). Thus it must be re-allocated using "micmem" type "B" after declaration and allocation using "paged" type "A".

Listing 22 Demo Program: for_micmem.pa

```
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define _USE_MIC
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {paged int[4]} A
$parray {micmem #A} B
$main{
    $create A(a)
    $malloc B(a)
    $for A(a) { *a=0; }
    $copy A(a) to B(a)
    $for k::B(a) { *a=k+1; }
    $copy B(a) to A(a)
    $for k::A(a) { printf("a[%d]=%d\n",k,*a); }
    $destroy A(a), B(a)
}
```

4 Thread Array Types

In PARRAY threads also form arrays. Programmers only need to learn a unified programming style to handle all kinds of threads. The key PARRAY command for parallelism is "`$for`", which may behave like a "for" command of other languages or a "map", depending on typing (also refer to Section 3.5). There are currently five supported thread types `mpi`, `pthd`, `omp`, `cuda` and `mic` for explicit thread creation. Contexts of "`$for`" commands can be nested. Any thread in a thread array may start an independent new array of threads. Figure 8 illustrates the nesting relations allowed by PARRAYbetween different thread types.

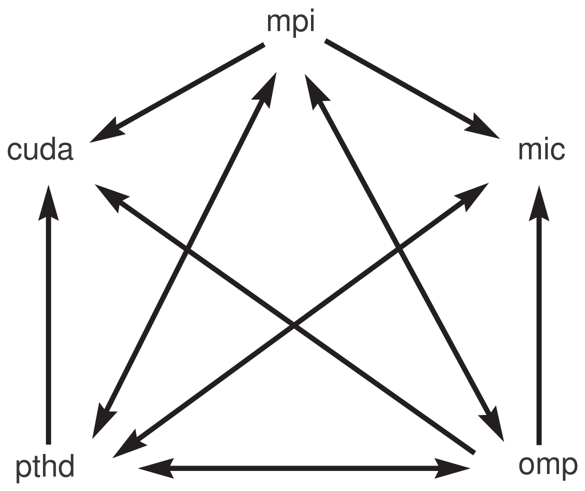

Figure 8 Allowed nesting relations of "for" commands indicated by arrows where mpi is multi-process distributed programming tool, pthd and `omp` are multicore programming tools, and mic indicates manycore OpenMP Offload programming tool.

4.1 Heterogeneous Parallelism: Tiling of Data Arrays (`tile/itile/otile`)

This paper studies the essence of heterogeneity from the perspective of language mechanism design. The proposed mechanism, called *tiling*, is a program construct that bridges two relative levels of computation: an outer level of source data in larger, slower or more distributed memory and an inner level of data blocks in smaller, faster or more localized memory。Block partitioning is often the starting point and the primary obligation of performance optimization. The inner memory is usually limited in size and often requires some particular shape and data layout to reach reasonable performance. For example, multiple computing steps may compete for the limited on-chip shared memory. Optimized allocation requires semantic understanding of several computing steps within a code — a particularly difficult task to the compiler if the average cache size per core is too small to ensure performance-transparent memory accesses.

The basic syntax of the "`for`" command for tiling is as follows:

`$for` *type_name1*`(`$data\_arg_1, data\_arg_2, \cdots, data\_arg_n$`)` `itile` *type_name2* { *code* }

$$\texttt{\$for}\ \textit{type\_name1}\texttt{(}\textit{data\_arg}_1, \textit{data\_arg}_2, \cdots, \textit{data\_arg}_n\texttt{)}\ \texttt{otile}\ \textit{type\_name2}\ \{\ \textit{code}\ \}$$

$$\texttt{\$for}\ \textit{type\_name1}\texttt{(}\textit{data\_arg}_1, \textit{data\_arg}_2, \cdots, \textit{data\_arg}_n\texttt{)}\ \texttt{iotile}\ \textit{type\_name2}\ \{\ \textit{code}\ \}$$

The keyword "`itile`" indicates loading a block from outer memory to the inner memory. The keyword "`otile`" indicates storing a block of inner memory to the outer memory. The keyword "`iotile`" merges both the keyword "`itile`" and the keyword "`iotile`".

Code Listing 23 illustrates the tiling matrix transposition on Intel MIC.

Listing 23 Demo Program: tiling.pa

```
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <immintrin.h>
#define _USE_MIC
$include "parray.pa"
int main(int argc, char *argv[]) { _pa_main(); return 0; }
$parray {micmem int [8][8]} A
$parray {micmem int [#A_1][#A_0]} B
$parray {vmem int [3][3]} T
$main{
    $create A(x,y)
    $for k::A(x,y) {*x=k; *y=-k;}
    $for A(x) itile T, B(y) otile T{
        $for TILE(x,y) {*y=*x;}
    }
    $for A_0(y) {
        $for A_1(y) {printf("%3d ",*y);}
        printf("\n");}
}
```

4.2 Launching Array of Threads (`for`)

In PARRAY, new threads are launched by the command "`$for`". Whenever such a command is executed, a new array of threads are created in runtime according to the given array type. If multiple threads are running such a command at the same time, multiple arrays of threads will be created. The basic command of launching an array of threads is as follows:

$$\texttt{\$for}\ \textit{thread\_array\_type}\ \texttt{as(}\textit{var}_1, \textit{var}_2, \cdots, \textit{var}_n\texttt{)}\ \{\ \textit{code}\ \}$$

where *thread_array_type* is an existing type declared in a "`$parray`" command or an anonymous type introduced on-spot. Each var_i is a variable in the context that is meant to be accessible by all created threads from their code body "*code*". Several "`$for`" commands can be nested in a static code block. For example, the main thread may launch an array of CPU threads, each of which then starts an independent array of MPI processes (see Code Listing 3).

4.3 Starting Multicore CPU Threads (pthd/omp)

Starting an array of CPU threads with anonymous "`pthd`" or "`omp`" thread array type requires a "`$for`" command:

`$for pthd` *dimtree* `as(`*var1*`,` *var2*`, ···) {`*code*`}` or
`$for omp` *dimtree* `as(`*var1*`,` *var2*`, ···) {`*code*`}.`

Implementation: The preprocessor will generate a separate C function that contains the code body of the "`$for`" command. Values of variables in the context are passed to the inner body as arguments in an "`as`" clause.

Example: Code Listing 24 illustrates how the main thread passes a local variable's value "`x`" to CPU threads through a formal argument "`as(int x)`". Commbjhand "`$tid`" returns the thread id. Note that thread type "`omp`" does not require passing variables, which is handled by the underlying compiler automatically.

Listing 24 Demo Program: pthd.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$main{
    int x=5;
    $for pthd[3][4] as(int x) {
        if ($tid==x) printf("tid=%d, tid_0=%d\n", $tid, $tid_0);
    }
}
```

4.4 Starting Manycore GPU Threads (cuda)

The following command launches an array of GPU threads:

`$for cuda` *dimtree* `as(`*var1*`,` *var2*`, ···) {`*code*`}.`

Implementation: PARRAY automatically regroups the number “n” of all threads into cuda blocks of size “_PA_DEFAULT_NTHREADS” where the re-definable C macro “_PA_DEFAULT_NTHREADS” in “_pa_cuda.h” indicates the default number 256 of threads in each cuda block. The number of blocks is the integer ceiling:

“(n+_PA_DEFAULT_NTHREADS-1)/_PA_DEFAULT_NTHREADS”.

The actual thread id will be checked in the generated cuda kernel to ensure that it is bound by “n”. The formal arguments will be the arguments of the kernel in the generated code. The low-level implementation will pass the argument data from the main memory to the shared memory of every multi-processor.

Example: Code Listing 25 shows a sub-program that copies an element at the position “$B[$tid]” of array “b” to the position “$A[$tid]” of array “a”. Line 6 of the program declares a GPU thread array type “C” that has “$size(A)/_PA_DEFAULT_NTHREADS” cuda blocks, each with 256 cuda threads. The formal arguments “a” and “b” are pointers with element types “$elm(A)” and “$elm(B)” respectively. This code can be used to perform the re-ordering of elements according to the given array types.

Listing 25 Demo Program: gpu_data_transfer.pa

```
 1  #include <stdio.h>
 2  #include <cuda.h>
 3  $include "parray.pa"
 4  int main(int argc, char *argv[]){_pa_main(); return 0;}
 5  $subprog foo(a, A, b, B)
 6      $for k::cuda[$size(A)] as($elm(A)* a, $elm(B)* b){
 7          a[$A[k]]=b[$B[k]]; }
 8  $end
 9  $parray {dmem float[4096][4096]} FROM
10  $parray {dmem float[#FROM_1][#FROM_0]} TO
11  $main{
12      _PA_INIT_GPU(0); // Initializing GPU0
13      $create FROM(from), TO(to)
14      $foo(to, $TO, from, $FROM)
15      $destroy FROM(from), TO(to)
16  }
```

If the preprocessor variable “grid” is 1 prior to the use of a cuda array type, then the type is expected to be two-dimensional fitting the grid and block structure of CUDA threads. Code Listing 42 illustrates the use of cuda thread array type.

4.5 Starting Manycore MIC Threads (`mic`)

Threads on MIC accelerators are similar to threads on CPUs with thread type `pthd`. The creator thread can be either a `pthd` thread on MIC or a special `mic` thread that is created by a CPU thread.

A `mic` thread can be created with the following command by a CPU thread/process:

`$for mic` *dimtree* `as(`*var1*`,` *var2*`,` ⋯`)` *code*

where variable arguments *var1*, *var2*, ⋯ are pointer variables used in CPU memory and MIC memory.

Vector arithmetics vary greatly in performance on Intel MIC (or Xeon Phi) manycore architecture. For example, double-precision intrinsic float-pointing divisions "`_mm512_div_pd`" on MIC only achieve about 1.5% the peak performance of Fused Multiply-Add (or FMA) vector instructions "`_mm512_fma_pd`".

Listing 26 Demo Program: micdiv.pa

```
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <immintrin.h>
$include "parray.pa"
#define _USE_MIC
#define N (1<<23)
#define nthreads (59*4)
$var nrepeats(32)
int main(int argc, char *argv[]){_pa_main(); return 0;}
$subprog MICDIV(code)
    $parray {vmem double[8]} VMEM
    $for k::omp[nthreads] {
    #ifdef __MIC__
        $create VMEM(a,b,c)
        for(int i=0; i<8 ; i++) {a[i]=1; b[i]=i; c[i]=k+1;}
        for (int n=0; n<N; n+=8) {
            $repeat(j,0,$nrepeats){
                for (int m=0; m<8; m++) code }}
        for (int i=0; i<8; i++) r+=c[i];
    #endif
    }
$end
$main{
    $for mic[1] {
        double Gflop = 1.0e-09*nthreads*N*$nrepeats;
        double r=0;
        _PA_CREATE_TIMER
        $MICDIV( c[m]=c[m]*a[m]+b[m]; ) // warmup
        _PA_TIME_NOW
        $MICDIV( c[m]=c[m]*a[m]+b[m]; )
        _PA_TIME_NOW
        printf("Fused Mul-Add: \t\t%.2Lf Gflops\n",
            Gflop*2/_PA_TIME_DIFF(1,2));
        $MICDIV( c[m]=a[m]/c[m]; )
        _PA_TIME_NOW
        printf("Division:\t\t%.2Lf Gflops\n",Gflop/_PA_TIME_DIFF(2,3));
    }
}
```

The code Listing 26 passes arithmetic code to a sub-program "MICDIV" as an argument. The sub-program performs the code repeatedly.

Implementation: For a 60-core MIC processor, we need 4 threads each on 59 cores, leaving one core left for the operating system. The code does not explicitly use Intel MIC intrinsics as the underlying compiler will vectorize the inner loop in this obvious case.

Listing 27 Result(MIC): micdiv

```
Fused Mul-Add:          965.02 Gflops
Division:               14.89 Gflops
```

4.6 Main Thread (main)

The PARRAY command "$main" defines the context of the main thread:

$main {*code*}.

In PARRAY we use the term "threads" for both CPU/GPU threads and cluster processes, unless it is specific for multi-server clusters where we adopt the term "processes".

The command "$main" essentially defines a single-CPU-thread-array context, which can be used to launch more CPU or GPU threads by "$for" commands in its code body. The generated code is located in a C function '_pa_main" that can be called by any user C code. The number "$size" (see Section 2.5) of threads is 1, and the only thread's id "$tid" is 0. The thread-array type is "pthd".

4.7 The Global Context (`global`)

The global context allows a code body to be placed in the global static environment of the generated C code with access to the current context. In code listing 28, the array type "`A`" is internal to the context of the sub-program "`foo`" which is usually not referable from the outside. The "`$global`" context inside of the sub-program, however, places a variable with initial value as the size of "`A`" in the global C environment, which is accessible from the main thread, although the sub-program is inserted in the code body of some CPU threads.

Listing 28 Demo Program: global.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
$include "parray.pa"
int main(int argc, char *argv[]){_pa_main(); return 0;}
$subprog foo()
    $parray {paged float[100]} A
    $global { int x=$size(A); }
$end
$main{
    $for pthd[4]{
        $foo(){}
    }
    printf("The size of A is %d inside the sub-program foo.\n", x);
}
```

4.8 Thread ID (`tid`)

The PARRAY command "`$tid`" returns the thread id of a thread in the current thread-array context. Such a context is defined by the current "`$for`" command. The immediate CPU, GPU and cluster thread array type is denoted by "`$tid(pthd)`", "`$tid(cuda)`" and "`$tid(mpi)`", respectively.

The expression "`$tid`" in the immediate code body of "`$main`" is always 0 as there is exactly one CPU thread in the main thread array. If multiple processes are launched then the expression returns 0 for every process, each as an independent thread array.

In general, the thread id expression uses the command: "`$tid`$\_d_1\_d_2\_\cdots\_d_n$(*thread_type*)" where d_i is a digit 0~9 indicating a sub-dimension (at most 10 sub-dimensions for each dimension) in the dimension tree. It returns the thread id for that specific dimension. If the thread-type ("`pthd`", "`cuda`" or "`mpi`") is the same as that of the current context, the brackets are omissible.

Implementation: Consider a CPU thread array type "`$parray {pthd[3][4]} P`". In the code body of "`$for P` {*code*}", the expression "`$tid_0`" returns "`i / 4 % 3`" that denotes the thread id of the column dimension, if this thread's overall id is "`i`". Code Listing 29 outputs "`tid=5, tid_0=1`" from thread No.5.

Listing 29 Demo Program: tid.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {pthd[3][4]} PTHD
$main{
    int i=5;
    $for PTHD as(int i) {
        if ($tid==i) printf("tid=%d, tid_0=%d\n", $tid, $tid_0);
    }
}
```

Implementation: For GPU thread arrays, thread id is derived from special integer structs "`blockIdx`" and "`threadIdx`" of CUDA. For a GPU thread array type "`$parray {cuda(...)[512] [[4][32]]} C`" (see Section 4.4), there are 512 cuda blocks in the grid and two thread dimensions (4*32) in each cuda block. The convention is that "`threadIdx.x`" denotes the thread id of the sub-dimension of size 32, while "`threadIdx.y`" denotes that of the sub-dimension of size 4. The grid may also split into two or three dimensions. The rightmost sub-dimension is always first scheduled. If the rightmost thread sub-dimension has size 32, it also forms a "warp". A warp is a SIMD unit in which all threads are synchronized at instruction-level, while different warps of a cuda block work like a memory-sharing SPMD unit and may execute different instructions of the same code kernel.

4.9 Thread Array Type of Current Context (`self`)

The PARRAY command "`$self(`*thread_type*`)`" or simply "`$self`" returns the name of the thread array type in the current context. For example "`$self(pthd)`" returns the immediate CPU thread array type. If the current context corresponds to a GPU thread array type within another CPU thread array type, the expression returns the name of the CPU thread array type. On the other hand, "`$self`" simply returns the thread array type in the immediate context. The type name of the "`$main`" context has thread type "`pthd`". The corresponding array type for "`$main`" is declared in the library file "`parray.pa`". The global context "`$global`" does not have thread array type and generates compilation error.

4.10 Synchronization of Threads (sync)

Synchronization command is to synchronize group-wise the threads of the current thread-array-type dimension that is identified with the dimension path "$\_d_1\_d_2\_\cdots\_d_n$":

$$\texttt{\$sync}\_d_1\_d_2\_\cdots\_d_n$$

where d_i is a digit 0~9 indicating a sub-dimension. This command should be executed on all threads that will be synchronized.

Example: Within the context of a GPU thread array type "C", the synchronization command "$sync_1" synchronizes all threads of the same row (*i.e.* the dimension "C_1"). Note that it only synchronizes the threads executing this command. It is the programmer's responsibility to ensure that all threads of a synchronized cuda block issues the same number of synchronization commands. Failure to observe this rule may result in deadlock.

Implementation: PARRAY currently only supports "$sync" for CPU and MPI thread arrays, and "$sync_1" for GPU inner-block synchronization. Other synchronization commands will be supported in future versions.

Code listing 30 illustrates how synchronization works for CPU threads.

Listing 30 Demo Program: sync.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
$include "parray.pa"
int main(int argc,char *argv[]){_pa_main(); return 0;}
$main{
    $for pthd[2] {
        if ($tid==0) {
            $sync
            printf("pthd thread %d begins.\n",$tid);}
        if ($tid==1) {
            sleep(5);
            printf("pthd thread %d ends.\n",$tid);
            $sync}}
}
```

4.11 The Parallel Context (parallel)

All for commands in a parallel context will run in parallel. A parallel context has the command:

`$parallel` {*code*}.

The calling thread of a "`$for`" command will create threads and wait for the return of all threads. Normally two PARRAY fors in the same context will inevitably be sequentialized by the calling thread. The parallel context, however, allows the calling thread to create threads of more-than-one thread array asynchronously and wait for their return together – essentially allowing all thread arrays within a parallel context running in parallel.

Implementation: The code generation of a "`$for`", in general, consists of four aspects: the launching code in the context calling the command, the starting code in the context of the for code body, the returning code of the context, and the waiting code in the calling context waiting for the returning of all threads. The generated code of a parallel context, on the other hand, collects the return-waiting codes of all "`$for`" commands and generate them together in the end. Other three parts of code as well as non-for commands of a parallel context are generated by the preprocessing compiler according to their original order in the context (possibly in different C functions though). The programmer needs to understand this simple mechanism in order to grasp the exact semantics of the parallel contexts.

Code listing 31 illustrates that the main thread initiates two thread arrays in parallel: the first as a CPU thread array with 3 threads and the second as an MPI process array with 2 processes. The beginning of the second thread array will start without waiting for the end of the first array. The code uses both libraries of pthread and mpi as the parallelism of both CPU multi-threading and clustering are present. The main C function also contains initialization and finalization of the MPI process array.

Listing 31 Demo Program: parallel.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <mpi.h>
#include <unistd.h>
$include "parray.pa"
int main(int argc,char *argv[]){
   MPI_Init(&argc, &argv);
   _pa_main();
   MPI_Finalize();
   return 0;
}
$main{
    $parallel{
        $for pthd[3] {
            printf("pthd thread %d begins.\n",$tid);
```

```
                sleep(5);
                printf("pthd thread %d ends.\n",$tid); }
            $for mpi[2] {
                printf("mpi process %d begins.\n",$tid);
                sleep(5);
                printf("mpi process %d ends.\n",$tid); }
        }
    }
```

The code in listing code:parallel requires 3 or more processes (including one for the main thread started by "`$main`") that interleave the begins and ends.

Listing 32 Result: mpirun -np 4 parallel

```
pthd thread 0 begins.
pthd thread 1 begins.
pthd thread 2 begins.
pthd thread 0 ends.
pthd thread 1 ends.
pthd thread 2 ends.
mpi process 0 begins.
mpi process 1 begins.
mpi process 0 ends.
mpi process 1 ends.
```

4.12 Starting MPI Process Arrays (`mpi`)

The most popular programming tool for clusters is Message-Passing Interface (or MPI). PARRAY generates MPI code when clustering parallelism is needed. There are a few differences though. MPI supports so-called Single Program Multiple Data (or SPMD). In this style of parallel programming, all processes in parallel execute the same program, while at any time, different processes may be reaching different points of the code. The behavioral differences of processes are programmed according to their processes IDs automatically assigned in linear order.

Cluster programming in PARRAY, on the other hand, has some distinct features. The first noticeable difference is that PARRAY processes form arrays. There can be multiple process arrays created and destroyed in runtime, but every process array must be started by some thread (the main thread, a CPU thread or another MPI process) using a "`$for`" command. Processes in a process array can communicate with each other collectively through distributed arrays (see Section 5.5).

Implementation: In a cluster environment, multiple processes can be started by command line "`mpirun`". There is a linear numbering for all mpi processes, though **pid 0 is reserved for the main thread started by the command**

"$main". All other threads are descendants of the main thread. Allocating a new MPI process array and releasing an existing process array are handled by all available processes collectively.

Partitioned Global-Address Space (or PGAS) communication is supported, if both header files "mpi.h" and "ga.h" are present. Not only different process arrays may exchange data through inter-communicators, it is also possible to support data transfer, for example, between a PGAS-distributed array (with global addresses) to an MPI-distributed array (without global addresses). Implemented combinations of data transfers can be found in library file "parray.pa".

To enable cluster parallelism, the header file "mpi.h" is needed. MPI must be initialized before and finalized after the main thread "_pa_main()". Code listing 31 starts from the main thread that initiates an array of 3 CPU threads, each of which in turn starts 2 MPI processes. The code requires a minimum of 3 processes including 1 control process and 2 MPI processes as a process array, and multiple process arrays are serialized, but launching more MPI processes will allow process arrays to run in parallel and achieve better performance.

Implementation: PARRAY in future will support other types of communications such as non-blocking message passing.

4.13 Send and Receive

The most basic message-passing is send and receive. The default scheme uses non-blocking Send and blocking Receive. Code listing 33 declares a source array type on one process with pid 1 within the process array and a target array type on one process with pid 3. The main thread starts 4 MPI processes and a message of 3 integers is sent from process 1 to process 3.

Listing 33 Demo Program: sendrecv.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <pthread.h>
$include "parray.pa"
int main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);
   _pa_main();
   MPI_Finalize();
   return 0;
}
$parray {paged int[3]} D
$parray {[mpi[disp(1)]][#D]} S
$parray {[mpi[disp(3)]][#D]} T
```

```
$main{
    $for k::mpi[4] {
        $create D(d)
        $for i::D(d) {*d=k;}
        $copy S(d) to T(d)
        if (k==3) $for i::D(d) {printf("%d ",*d);}
        $destroy D(d)
    }
}
```

4.14 Alltoall Communication

The well-known "MPI_Alltoall" communication is to exchange 2D arrays' rows among all processes so that the j-th row of the i-th process goes to the i-th row of the j-th process. If the source and the target arrays are represented as distributed arrays, then such data exchange becomes the swapping between a thread array dimension and the column dimension of the 2D arrays. Code listing 34 illustrates this communication pattern.

Listing 34 Demo Program: alltoall.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
$include "parray.pa"
int main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);
   _pa_main();
   MPI_Finalize();
   return 0;
}
$parray {mpi[3]} M
$parray {paged int[3][3]} D
$parray {[[#M][#D_0]][#D_1]} S
$parray {[[#D_0][#M]][#D_1]} T
$main{
    $for k::M {
        $create D(s,t)
        $for D(s) {*s=k;}
        $copy S(s) to T(t)
        if (k==1) $for D(t) {printf("%d ",*t);}
        $destroy D(s,t)
    }
}
```

4.15 Scatter Communication

The well-known “`MPI_Scatter`” communication dispatches the rows of a 2D array on process 0 to different processes so that the i-th row of process 0 is copied to the row buffer of process i.

If the 2D source array is entirely local (*e.g.* with memory type `paged`), it is assumed to be on process 0. Other scattering patterns with different source processes require adding a displaced distribution dimension like Code Listing 33.

Code listing 35 illustrates this communication pattern.

Listing 35 Demo Program: scatter.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <pthread.h>
$include "parray.pa"
int main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);
   _pa_main();
   MPI_Finalize();
   return 0;
}
$parray {mpi[3]} M
$parray {paged int[3][3]} S
$parray {[#M][#S_1]} T
$main{
    $for k::M {
        $create S(s), S_1(t)
        if (k==0) $for i::S(s) {*s=i;}
        $copy S(s) to T(t)
        if (k==1) $for i::S_1(t) {printf("%d ",*t);}
        $destroy S(s), S_1(t)
    }
}
```

4.16 Mixtomix Communication

PARRAY allows collective communication patterns not corresponding to any typical MPI patterns. For example, the resulting communication may require transferring data segments noncontiguously between each pair of processes. Code listing 36 illustrates such a communication pattern.

Listing 36 Demo Program: mixtomix.pa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
$include "parray.pa"
int main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);
   _pa_main();
   MPI_Finalize();
   return 0;
}
$parray {mpi[2][2]} M
$parray {paged int[2][2]} D
$parray {[#M][#D]} S
$parray {[[#M_1][#D_0][#M_0]][#D_1]} T

$main{
    $for k::M {
        $create D(s,t)
        $for D(s,t) {*s=k;*t=-1;}
        $copy S(s) to T(t)
        if (k==2) $for D(t) {printf("%d ",*t);}
        $destroy D(s,t)
    }
}
```

5 Advanced Array Types

More notations of array typing are required for various applications. As a form of representational completeness, it can be shown that any indexing expression is producible from PARRAY types, as long as the expression consists of only integer expressions, multiplication, division and modulo operators, additions and compositions between expressions.

5.1 Advanced Use of Array Types

Dimension references may form more sophisticated patterns, whose exact semantics observe some predefined rules.

Example: Consider a simple regular array type "`$parray {pinned float[[3][3]][2]} A`" and another type "`B`" that refers to "`A`" with the two column sub-dimensions swapped:

```
$parray {dmem float[[#A_0_1][#A_0_0]][#A_1]} B.
```

More sophisticated types can be derived from these simple ones. Type references may appear to a dimension and some of its descendent sub-dimensions, for example:

```
$parray {dmem float[[3][#A_1]][2]#B} C.
```

The offset of "`C_0`" involves the offsets of both "`A_1`" as a sub-dimension and "`B`" as a sup-dimension. The offset of every dimension, in general, can be viewed as two parts: the regular-part offset that is related to the dimension's position in the entire dimension tree and the fixed-part offset that is specific to that dimension.

The left column sub-dimension "`C_0_0`" is entirely **regular**, and its offset is transformed by the root reference "`#B`" such that "`$C_0_0[i]==$B[(i%3)*2*2]`" where the modulo operator "`i%3`" ensures the index being in scope, and multiplication "`*2*2`" represents a regular gap of 4 between consecutive indices.

The right column sub-dimension "`C_0_1`" is **fixed** on its external reference to "`A_1`". That means the sub-dimension's relative position in the dimension tree and the root reference have no influence on its offset: "`$C_0_1[i]==$A_1[i]`".

Sometimes a dimension may be partly regular and partly fixed. In that case, the relative position and the type references above the dimension will only affect the regular part. As an example, the dimension "`C_0`" satisfies:

```
$C_0[i] == $B[i/2%3*2*2] + $A_1[i%2]
```

where the expression "`i/2%3`" denotes the index `i`'s projection onto the left column sub-dimension "`C_0_0`", and the expression "`i%2`" denotes the projection onto "`C_0_1`".

The overall rule that a user needs to remember is that whenever there is a type reference, the offset of that dimension will be fixed.

5.2 Dimension Contiguity Expression (`cnt/stp`)

The performance of data transfer is not only related to bandwidth and latency of the channel but also the granularity of the contiguous segments. For example, memory-copying a thousand 10KB contiguous data blocks across PCI to GPU will be dozens of times slower than copying 10MB in one contiguous block. It is thus important to analyze this at compile-time for code generation.

A type "`A`" is contiguous if its offsets satisfy the condition: "`$A[i]==i`" for "`i<$size(A)`". The boolean PARRAY expression

`$cnt(`*type_name*`)`

checks whether the type "*type_name*" is contiguous. A related PARRAY expression

`$stp(`*type_name*`)`

returns the step of contiguity (*i.e.* the regular gap between consecutive indices) if the type is indeed contiguous:

`$A[i] == i*$stp(`*type_name*`)`

where "`i<$size(A)`"; if it is deemed noncontiguous, the expression's value is 0. Thus "`$cnt(`*type_name*`)`" is the same as the boolean expression "`($stp(`*type_name*`)==1)`".

Example: Consider an array type "`A`" with three dimensions:

```
$parray {paged float [[3][3]][2]} A
```

According to Section 2.3, the offset of "`A`" satisfies "`$A[i]==i`", and therefore the type is contiguous. Another array type "`B`" refers to the dimensions of "`A`":

```
$parray {paged float [[#A_0_1][#A_0_0]][#A_1]} B
```

have the column sub-dimensions swapped. Then the type (*i.e.* the root dimension) "`B`" is no longer contiguous, but the row dimension "`B_1`" still is. That means the data transfer from *B* to *A* can be performed with a number of "`memcpy`" commands for contiguous memory copy of "`$size(B_1)`" elements. For reasonably large segments, the C command `memcpy` significantly outperforms element-wise copying in a loop. The contiguous step expressions satisfy: "`$stp(A)==1`", "`$stp(A_1)==1`", "`$stp(A_0)==2`", "`$stp(A_0_1)==2`", and "`$stp(A_0_0)==6`", while "`$stp(B)==0`", "`$stp(B_0)==0`" and "`$stp(B_0_1)==6`".

5.3 Array Types with Parameters

The simple definition of a PARRAY type is fairly similar to C macros. The size of a dimension may be a variable or even some expression. However, the variable names of the expressions in a dimension tree are fixed and subject to naming conflicts in a large code. By identifying the variables as formal parameters, it is then possible to work with multiple instances of the same array type in a context.

Another issue is that the runtime values of the involved variables may not be accessible by the created threads in runtime. Section 5.4 illustrates the means to pass arguments from the caller thread to the callee threads. It is advantageous to identify the involved variables and pass their values among arguments so that these variables become accessible to the local syntactical context and allow correct computation of the dimension sizes and offsets by the callee threads.The following command declares an array type with parameters:

$$\texttt{\$parray}\textit{<type}_1\ \textit{var}_1, \textit{type}_2\ \textit{var}_2, \cdots, \textit{type}_n\ \textit{var}_n\textit{>}\ \{\textit{mtt element\_type dimension\_tree}\}\ \textit{type\_name}.$$

Each var_i is the variable name of a parameter with a C data type $type_i$. The type is only used when passing the parameters as arguments (see Section 5.4).

Example: Code Listing 37 first defines a 6x6 two-dimensional array and a 2x4 window with displacement "`(y,x)`" as parameters. By setting different parameters, the window type gives rise to multiple type instances in the same context. In this example, the data in window "(1:2,2:5)" is copied to window "(4:5,1:4)" (see Figure 9).

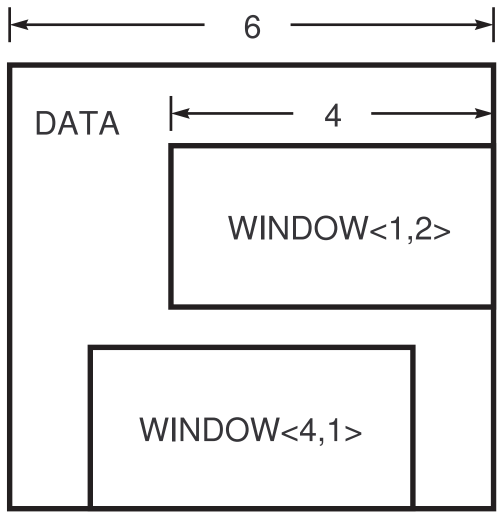

Figure 9 Copying from "`WINDOW<1,2>`" to "`WINDOW<4,1>`"

Listing 37 Demo Program: parameters.pa

```
#include <stdio.h>
#include <stdlib.h>
$include "parray.pa"
int main(int argc,char *argv[]){_pa_main(); return 0;}
$parray {paged int[6][6]} DATA
$parray<int y, int x> {paged int[2#disp(y)#DATA_0][4#disp(x)#DATA_1]}
    WINDOW
$main{
    $create DATA(d)
    $for i::DATA(d) {*d=i;}
    $copy WINDOW<1,2>(d) to WINDOW<4,1>(d)
    $for DATA_0(d) {
        $for DATA_1(d) {printf("%2d ",*d);}
        printf("\n");}
}
```

Listing 38 Result: parameters.pa

```
 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  8  9 10 11 29
30 14 15 16 17 35
```

5.4 Advanced Handling of Parameters (args/vars/para)

The PARRAY command "`$args(`*type_name*`)`" returns the list of formal parameters with C types, while "`$vars(`*type_name*`)`" returns the parameter variable list without C types. For example, a type definition like

`$parray<int n, int m> {smem float[n][m]} R`

describes a two-dimensional array type in GPU shared memory with arguments "`n`" and "`m`". The list "`$args(R)`" of parameters with types yields "`int n, int m`", while the parameter variable list "`$vars(R)`" returns "`n,m`".

The access to an array type can carry actual parameters, which will be used to replace the the formal arguments during compilation. The complete syntax to refer to an array type is as follows:

array_name`.`*ext*<$para_1, para_2, \cdots, para_n$> `on` <*mtt*>

where *ext* is a name extension, each $para_i$ is a user-defined parameter, and *mtt* is a memory/thread type. The extension will add prefix "`_pa_`*ext*`_`" to a parameter variable. This is useful when the parameters of two types are passed as functional arguments and potentially cause name conflict. The PARRAY command "`$para(`*type_name*`)`" returns the actual parameters of the given type. For a distributed array type with multiple MTTs, *mtt* specifies the user-requested MTT combination (see Sectionsec:dsitrib).

Example: Code Listing 39 illustrates the handling of parameters. As GPU threads cannot access the global host memory directly and the variables in it, if such variables are used in the definition of array types involved, their values must be passed to the thread's code through functional arguments. If the variables of more-than-one type are passed at the same time, certain renaming with extensions is required.

Listing 39 Demo Program: args.pa

```
#include <stdio.h>
#include <cuda.h>
$include "parray.pa"
int main(int argc, char *argv[]){_pa_main(); return 0;}
$subprog foo(a, A, b, B)
    $parray {cuda [$size(A)]} C
    $for k::C as($elm(A)* a, $tparas(A), $elm(B)* b, $tparas(B)){
        a[$A[k]]=b[$B[k]]; }
$end
$parray<int n> {dmem float[n][n]} FROM
$parray<int n> {dmem float[#FROM_1<n>][#FROM_0<n>]} TO
$main{
    int size=4096;
    _PA_INIT_GPU(0); // Initializing GPU No.0
    $create FROM<size>(from), TO<size>(to)
    $foo(to, $TO<size>, from, $FROM<size>)
    $destroy FROM(from), TO(to)
}
```

5.5 **Distributed Arrays** (on)

A distributed array type does not have an overall memory/thread type (MTT) but consists of dimensions mixed of thread array type and memory array type. The rule is that the MTT of an outer dimension dominates those of its inner sub-dimensions (whose MTTs ignored). Currently no-more-than two MTTs are supported: one thread array type and one memory array type. Unsupported distributed array types are allowed for type-checking purposes but not valid for "`$copy`".

Implementation: For a distributed array type "`foo`" with thread type "`PTHD`" and memory type "`DMEM`", macros will depend on an MTT argument.

Example: Code Listing 40 firsts declares a CPU thread array type "`PTHD`" and a device-memory array type "`DMEM`". Then the distributed data array type "`DATA`" is declared with column dimension referring to "`PTHD`" and the row dimension referring to "`DMEM`". The code outputs 1 and 2 as the offsets.

Listing 40 Demo Program: distrib_array.pa

```
#include <stdio.h>
$include "parray.pa"
```

```
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {pthd[4]} PTHD
$parray {dmem float[4096]} DMEM
$parray {[#PTHD][#DMEM]} DATA
$main{
    printf("%d %d\n", $DATA on<_PA_THREADMT>[4098],
    $DATA on<_PA_LOCALMT>[4098]);
}
```

Example: Code Listing 41 first declares CPU thread array type "PTHD" and a host-memory array type "HMEM". Then the distributed data array type "DATA" is declared with column dimension referring to "PTHD" and the row dimension referring to the row dimension "HMEM_1". The CPU thread array distributes each row of a "HMEM" array to a local array in a thread in parallel.

Listing 41 Demo Program: distrib_data_transfer

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
$include "parray.pa"
int main(int argc, char *argv[]){ _pa_main(); return 0; }
$parray {pthd[4]} PTHD
$parray {paged int[4][4096]} HMEM
$parray {[#PTHD][#HMEM_1]} DATA
int *data;
$main{
    $malloc HMEM(data)
    $for i::HMEM_0(data) {*data=i;}
    $for k::PTHD {
        $create HMEM_1(ldata)
        $copy HMEM(data) to DATA(ldata)
        printf("%d on thread %d.\n", k, ldata[0]);
    }
}
```

6 Sub-Programming And Conditional Compilation

A sub-program is like a general C macro function. Array types and even C code can be passed as arguments. When passing an array type through a sub-program's argument, certain name conversion is performed so that a sub-program can be included multiple times in the same context without causing name conflicts for the generated macros.

6.1 Sub-Programs (`subprog/insert`)

A sub-program in PARRAY has the following command:

`$subprog` *name*`(`*formal_arg1*`,` *formal_arg2*`,` $\cdots$ `,` *formal_arg n*`)` *code* `$end`

with a number of formal arguments and a code body. Definitions of sub-programs must not be nested. Inserting a sub-program requires another PARRAY command:

`$insert` *name*`(`*actual_arg1*`,` *actual_arg2*`,` $\cdots$ `,` *actual_arg n*`)` { *actual_code_arg* }.

A sub-program works like a C macro. That means on every insertion, the code body syntactically substitutes the insertion command after the formal arguments of the sub-program are replaced by the actual arguments of the insertion command. Any word prefixed with some formal argument followed by underline "_" and some suffix will have its prefix replaced by the corresponding actual argument. The insertion of a sub-program has an alternative simpler syntax as a `$`-expression:

`$`*name*`(`*formal_arg1*`,` *formal_arg2*`)`.

The simple syntax is convenient when the inserted sub-program code is an expression and does not contain actual code argument.

The inserted code body *actual_code_body* is optional and acts as the last or n+1-th actual argument. When it is empty though, there are only n actual arguments, which must match the number of formal arguments of the inserted sub-program. The inserted code body, to be parsed in the inserted context, has a subtle distinction from other actual arguments that are parsed in the inserting context.

The insertion of sub-programs can be recursive. This is useful when a large-scale problem is decomposed to different levels of parallelism. It is the responsibility of the programmer to ensure the number of insertion to be finite during compilation; otherwise a compilation error is generated.

A sub-program creates its own lexical context in which array types declared locally are not referable from other contexts unless they are passed through the arguments on insertion. This also means that different contexts allow PARRAY types of the same name without any name conflict. However, this rule does not apply to variable names that are handled by the C compiler.

Implementation: A sub-program will be re-compiled on every insertion. The actual generated code depends not only on the code body but also on the contextual environment of the insertion. This mechanism should not be confused with function invocation in C where the invoked function can be pre-compiled and linked for different software packages.

Contextual distinction is achieved by the compiler automatically adding a unique prefix to every type name. That means the code in one context cannot refer to the type declared in another context (from a different sub-program or the same sub-program's different insertion instances). However such prefix will only be added once. A type name passed through the argument to a sub-program will retain its original prefix so that the code body of that sub-program can refer to the external type through its formal argument. If an array type name is passed as a non-code actual argument to a sub-program, it is automatically augmented with a prefix from the inserting context.

Example: Code Listing 42 illustrates a PARRAY sub-program that computes "c+=a*b" for two input arrays "a" of type "A" and "b" of type "B" and the output array "c" of type "C". The sub-program first checks that all three array types are indeed two-dimensional. The computing task is divided over the rows of "a" among available threads in the current thread-array context. Thus the actual generated code depends on the context in which the sub-program is inserted. The first insertion is placed in a CPU thread array's for and uses 5 threads, while the second insertion launches a GPU kernel with 16 cuda blocks and 128 threads in each block. Pointers must be passed as arguments to the GPU code. In this code, the size of the matrix is a constant. It can be a variable whose value is determined in runtime and then passed as an argument to the code body of the CUDA thread array. Copying data from and to the GPU device memory requires insertion of the general "_PA_Copy" sub-program, which is pre-defined in the library file "parray.pa" of basic sub-programs.

Listing 42 Demo Program: subprog.pa

```
#include <stdio.h>
#include <pthread.h>
#include <cuda.h>
$include "parray.pa"
int main(int argc, char *argv[]){_pa_main(); return 0;}
#define N 32
$subprog gemm(a, A, b, B, c, C)
    $if ($ndim(A)==2 && $ndim(B)==2 && $ndim(C)==2){
        $parray {[$size][$size(C_0)/$size]#C_0} CP
        int o=$CP_0[$tid];
        $for CP_1(c,a) {
            $for C_1(c),B_1(b) {
                $for A_1(a),B_0(b){
                    (c[o]) += (a[o]) * (*b);}}}
    }
$end
$parray {paged int[[N/16][N/16]][[16][16]]} DAT
$parray {paged int[[#DAT_0_0][#DAT_1_0]][[#DAT_0_1][#DAT_1_1]]} MAT
```

```
$parray {dmem int #DAT} DMEM
int *a,*b,*c1,*c2;
$main{
    $malloc DAT(a,b,c1,c2)
    $for DAT(a,b,c1,c2) { *a=rand()%10; *b=rand()%10; *c1=0; *c2=0; }
    $for pthd[4] {
        $gemm(a,$MAT,b,$MAT,c1,$MAT)
    }
    _PA_INIT_GPU(0);
    $create DMEM(da,db,dc)
    $copy DAT(a),DAT(b),DAT(c2) to DMEM(da),DMEM(db),DMEM(dc)
    $for cuda[2][8] as(int* da, int* db, int* dc){
        $gemm(da,$MAT,db,$MAT,dc,$MAT)
    }
    $copy DMEM(dc) to DAT(c2)
    $for DAT(c1,c2) { if ((*c1)!=(*c2)) {printf("ERROR\n"); exit(1);} }
    $destroy DMEM(da,db,dc), DAT(a,b,c1,c2)
}
```

6.2 PARRAYIncluded Library (include)

A PARRAY library is a file containing PARRAY code. The command to incorporate a PARRAY library has the following command:

`$include` ''*filename*''

where *filename* is the name of the library file. The library file is included in the program text during PARRAY preprocessing. That means the PARRAY commands in a library file are parsed and compiled. The similar C directive command "`#include`", however, is a pure C notation and will be ignored by the preprocessor and only processed during C compilation. As a convention, a PARRAY library usually has suffix extension "`.pa`".

The inclusion of the system-pre-defined library "`parray.pa`" is obligatory before the C's main function. It contains the basic array types and basic sub-programs such as "`_PA_Copy`". User's own libraries can be included anywhere in a program.

6.3 Preprocessed Expressions (eval)

During preprocessing, PARRAY can evaluate some integer expressions with the command:

`$eval(`*parray_exp*`)`.

To be able to evaluate such an expression successfully during preprocess, all values involved must be either constants or preprocessed variables. C or C++ macros are not recognized by PARRAY preprocessor. If the evaluation is unsuccessful, the original expression will be left there intact as text in the code.

6.4 Conditional Preprocessing Compilation (`if`)

There are various static conditions that can be checked during preprocessing. The command for conditional preprocessing compilation has the following command:

`$if (`*parray_exp*`) {` *code_if* `} $else {` *code_else* `}`.

This command checks whether the type condition "*parray_exp*" during preprocessing compilation. If the condition can be statically evaluated and is true, the code of "*code_if*" will be generated; if false, "*code_else*" is generated instead. If "*code_else*" is empty, "`$else{}`" becomes omissible. Code Listing 42 has used this command to ensure that all argument array types of a sub-program have exactly two dimensions.

If, however, the type condition "*parray_exp*" cannot be statically evaluated, the command will become the "`if...else ...`" command of C.

6.5 Preprocessed Variables (`var`)

Untyped variables can be introduced in processing compilation during which declaration and value assignment will be performed before the compilation of the generated C or C++ code. Declaration and assignment of a preprocessed variable uses the following command:

`$var` *variable_name*`(`*parray_exp*`)`

where preprocessed value of "*parray_exp*" will be assigned to the variable "*variable_name*" as an initial value for declaration or an updated value for assignment. If the value can be determined during preprocessing, then the value assigned to the variable is the result of evaluation; otherwise, the expression will be assignment to the variable as program text.

Implementation: Preprocessed variables are often used in PARRAY as parameters to the next PARRAY commands. The roles of these variables range from dictating synchrony and buffering of message passing to extra arguments required for file image "`mmap`". For example, the command

```
$var file(fd)
```

in Code Listing 16 sets the file handler before the allocation of any "`mmap`" array. Code Listing 43 uses a parameter:

```
$var grid(1)
```

to indicate that the following "`cuda`" array type must be compiled to map GPU's block-thread hardware setup directly instead of treating the type declaration as a one-dimensional array of threads with block size set as default

(256). By switching between 0 and 1 for the variable "`grid`", one can see the performance difference. **Note that a preprocessed variable is usually reset to its default value by the command that uses it. That means it must be set to a non-default value every time whenever this is required.**

6.6 Checking Emptiness (`blank`)

It is often useful to be able to check whether a given argument of a sub-program is empty. Recursive insertion of sub-programs may use such checking as the terminal case of conditional compilation. This condition can be checked with the command:

`$blank(`*text*`).`

Its value is 1 if the text is indeed empty or 0 otherwise.

6.7 Repeated Code Generation (`repeat`)

Syntactically repeated code generation is useful when the repetition causes unwanted performance overheads or the generated code involves an unfixed number of arguments during preprocessing compilation. Code repetition uses the following command:

`$repeat (`*index_var*`, `*start_index*`, `*range_limit*`) { `*code*` }`

where *index_var* is assigned with *start_index* first and repeated until *range_limit*-1. The code body can access the repeat index variable using "`@`*index_var*`@`". Code Listing 47 has used this command for generating inner-loop code.

7 Case Studies

In this section, we will show a few examples of GPU, clustering and MIC programming with advanced performance optimization. The aim is to illustrate the potential of PARRAY code performance. Some algorithmic and coding techniques may require architectural understanding of hardware devices in depth. PARRAY is intended to be a programming tool that is easy to learn from the start and still allows the programmer to achieve deep levels of performance optimization within the programming style without the need to use lower-level programming interfaces.

7.1 GPU SGEMM

Single-precision matrix multiplication or sgemm is one of the most basic routines of scientific computing. This example is included to illustrate a wide range of commands and notation from previous sections. Code Listing 43 contains a GPU sgemm code that is an improved version from CUBLAS 2.0. The code works well on all major models of NVIDIA GPUs including the GT200 series and Fermi, does not use textile or constant memory, and does not depend on binary-level tuning. It can well be the fastest CUDA-based code of this kind.

An additional advantage of implementation as PARRAY sub-program is that the array type that describes array's memory layout can become the arguments of the sub-program. That means if the arrays are not as regularly ordered in memory as a row-major regular array, the code still works correctly. In fact, if the input arrays have contiguous element layout up to a certain extent (*e.g.* for every consecutive 16 elements), the performance of the code will be fairly close to its peak. Unlike putting specific array orientation as arguments in BLAS, PARRAY allows all such information concentrated in the type argument. Such flexibility allows a single code to work for various data layouts.

Listing 43 Demo Program: sgemm.pa

```
#include <stdio.h>
#include <cuda.h>
#include <cublas.h>
$include "parray.pa"
#define N 4096

int main(int argc,char *argv[]) {_pa_main(); return 0;}

$subprog sgemm(a, b, c, type)
    //$var grid(1)
    $parray {cuda [[N/16][N/256]][[4][32]]} CUDA
    $parray {dmem float[[N/16][[4][4]]#type_0][[N/32][32]#type_1]} A
    $parray {dmem float[[N/32][32]#type_0][[N/256][[128][2]]#type_1]} B
    $parray {dmem float[#A_0][#B_1]} C
```

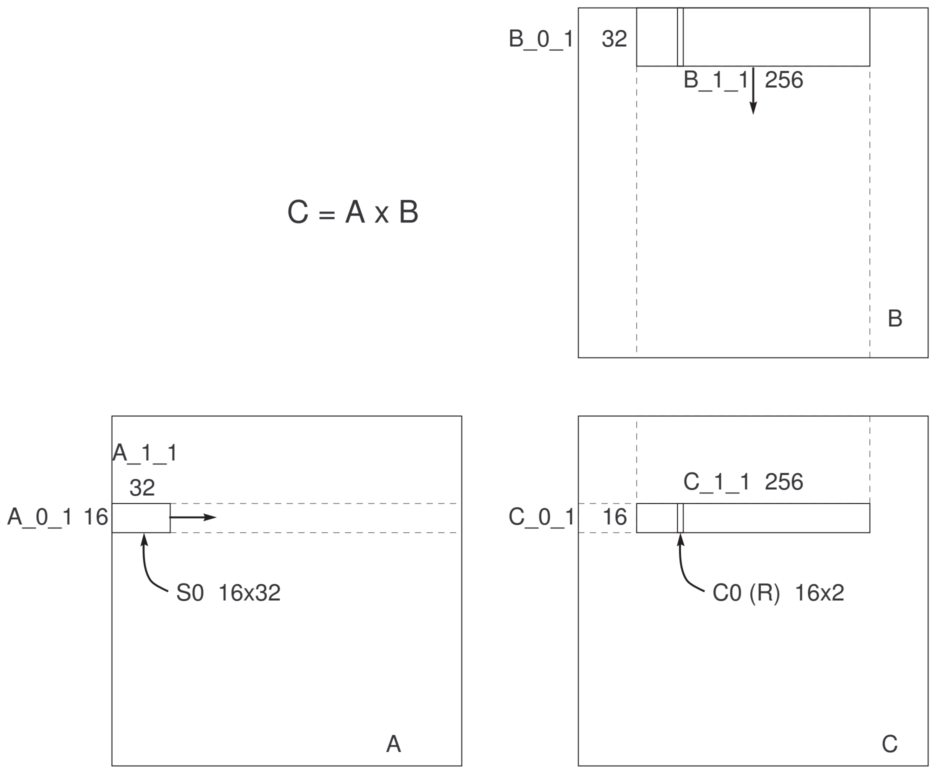

Figure 10 Matrix Multiplication

```
    $parray {dmem float[#C_0_1][#C_1_1_1]} C0
    $parray {smem float[32][17]} S        // avoid bank conflict
    $parray {smem float[32#S_0][[4][4]#S_1]} S0
    $parray {rmem float[16][2]} R

    $for CUDA as(float* a, float* b, float* c){
        $create S(s), R(r)
        $for R(r) {*r=0;}
        a += $A[[$tid_0_0][[0][$tid_1_0]]] [[0][$tid_1_1]];
        b += $B[0] [[$tid_0_1][[$tid_1][0]]];
        c += $C[[$tid_0_0][0]] [[$tid_0_1][[$tid_1][0]]];
        for (int i=0; i<$size(A_1_0); i++){
            $copy A_0_1_0(a+$A_1_0[i]) to
                S0_1_0(s+$S0[$tid_1_1][[0][$tid_1_0]])
            $sync_1
            #pragma unroll
            for (int m=0; m<$size(A_1_1); m++){
                float2 b2;
                // better memory bandwidth for float2 on GT200
                float* b1= b+$B_0[i][m];
                if ($cnt(B_1_1_1)) b2 = ((float2*)b1)[0];
                else {b2.x = b1[0]; b2.y = b1[$B_1_1_1[1]];}
                #pragma unroll
                for (int j=0; j<16; j++){
                    r[$R[j][0]] += b2.x*s[$S[m][j]];
                    r[$R[j][1]] += b2.y*s[$S[m][j]]; }}
            $sync_1
        }
        $copy R(r) to C0(c)
    }
$end

$parray {paged float[N][N]} HT
$parray {paged float[#HT_1][#HT_0]} HRT
$parray {dmem float#HT} DT
$main{
    _PA_INIT_GPU(0);
    cublasInit();
    $create HT(ha,hb,hc,ref), DT(da,db,dc)
    $for HT(ha,hb) {
        (*ha)=rand()/(float)RAND_MAX; (*hb)=rand()/(float)RAND_MAX;}
    printf("\nCUBLAS sgemm: ");
    $copy HT(ha),HT(hb) to DT(da),DT(db)
    cublasSgemm('t', 't', N, N, N, 1.0f, da, N, db, N, 0.0f, dc, N);
    // warm up
    cudaThreadSynchronize();
    _PA_CREATE_TIMER
    cublasSgemm('t', 't', N, N, N, 1.0f, da, N, db, N, 0.0f, dc, N);
    // test
    cudaThreadSynchronize();
    _PA_TIME_NOW
    printf("%Lf GFlops\n",
        (long double)2*N*N*N/1000/1000/1000 / _PA_TIME_DIFF(0,1));
    $copy DT(dc) to HT(hc)
    $copy HRT(hc) to HT(ref)  // transposition
    printf("PARRAY sgemm: ");
    $copy HT(ha) to DT(dc)  // mess up the result space
    $copy HT(ha),HT(hb) to DT(da),DT(db)
    $sgemm(da, db, dc, $DT) // warm up
    _PA_TIME_NOW
    $sgemm(da, db, dc, $DT) // test
    _PA_TIME_NOW
    $copy DT(dc) to HT(hc)
    printf("%Lf GFlops\n",
        (long double)2*N*N*N/1000/1000/1000 / _PA_TIME_DIFF(2,3));
    printf("Comparison: %s.\n",
        (CompareL2fe(hc,ref,N*N,1e-6f)?"PASS":"FAIL"));
    $destroy HT(ha,hb,hc,ref), DT(da,db,dc)
}
```

Running on a Fermi M2070 with CUDA 4.0 (on Tianhe-1A) presents the following results.

Listing 44 Result(Fermi): sgemm

```
CUBLAS sgemm: 609.527 GFlops
PARRAY sgemm: 613.262 GFlops
Comparison: PASSED.
```

The code performs comparably better on a GTX285:

Listing 45 Result(GTX285): sgemm

```
CUBLAS sgemm: 420.310 GFlops
PARRAY sgemm: 444.028 GFlops
Comparison: PASSED.
```

The way that the sgemm code works is to partition the input arrays “a” in type “A” and “b” in type “B” according to the memory size of every cuda block with 128 threads. Each thread stores 16x2 floats from the register file of “r” in type “R” to result array “c”. The threads of each cuda block first load totally 16x32 floats of array “a” in 4 steps to the shared memory “s” in type “S0” that before a block-wise synchronization among threads. Each thread then reads two row-wise consecutive floats and computes their multiplication with the part of “a” in the shared memory. The result becomes the update to the part of “c” in the register files of the threads. Such computation is repeated by sliding the rows of “b” downwards and the parts of “a” in shared memory rightwards until all relevant elements of the input arrays for the cuda block’s part of “c” are processed. Figure 10 illustrates the partitioning of the array types.

7.2 GPU-Cluster FFT

For *small-scale* FFTs whose data are held entirely on a GPU device, their computation benefits from the high device-memory bandwidth. This conforms to an application scenario where the main data are located on dmem, and FFT is performed many times. Then the overheads of PCI transfers between hmem and dmem are overwhelmed by the computation time.

If the data size is too large for a GPU device or must be transferred from/to dmem every time that FFT is performed, then the PCI bandwidth becomes a bottleneck. The time to compute FFT on a GPU will likely be overwhelmed by data transfers via PCIs. This is the scenario for large-scale FFTs on a GPU cluster where all the data are moved around the entire cluster and between hmem and dmem on every node. The performance bottleneck for a GPU cluster will likely be either the PCI between hmem and dmem or the network between nodes — whichever has the narrower bandwidth.

The 3D PKUFFT algorithm first distributes a 3D array with dimensions Z, Y and X along dimension Z. Every computing node holds N/P 2D planes for Y and X dimensions where N denotes the size of each dimension and P the number of GPUs. Every 2D plane is transferred to the GPU for 2D FFT computation (using the existing library CUFFT) and then transferred back to the main memory. All computing nodes then perform an Alltoall-like collective communication to aggregate the Z dimension on each computing node and redistribute the array along dimension Y. Data are then computed for 1D FFT and sent back to the main memory.

A major operation of the algorithm requires transposing the entire array, which usually involves main-memory transposition within every node and Alltoall cluster communication. The main optimization of this algorithm is to re-arrange and decompose the operation into small-scale GPU-accelerated transposition, large-scale Alltoall communication and middle-scale data-displacement adjustment that is performed during communications. Then the main-memory transposition is no longer needed! The price paid is to use a non-standard Alltoall with non-contiguous process-to-process communications.

This algorithm performs dimensional adjustment during collective communication so that the changes of offsets from the source array to the target array have, to some extent, completed part of the transposition for free and no longer require main-memory copying. That means the large-scale array transposition is achieved by exchanging data between all computing nodes, while the medium-scale transposition is achieved by non-contiguously transmitting multiple segments between each pair of computing nodes. The small-scale transposition (still required for maintaining granularity of network communication) is left to the GPU to perform before and after GPU-based FFTs.

Listing 46 Demo Program: pkufft.pa

```
#include <stdio.h>
#include <cuda.h>
#include <mpi.h>
#include <cufft.h>
$include "parray.pa"
int main(int argc,char *argv[]){
   MPI_Init(&argc, &argv);
   _pa_main();
   MPI_Finalize();
   return 0;
}
#define N 128
#define P 4
float2 *host, *hbuf, *dev, *data1, *data2;
$parray {mpi[P]} PROCS
$parray {pinned float2[P][[N/P][[N][N]]]} HSTS
$parray {dmem float2[N][[N][N]]} DEVS
$parray {[#PROCS][#HSTS_1]} SCATTER
$subprog fft(HST, DEV, PROCS) {
    $parray {[#DEV_1][#DEV_0]} DEVT
    $parray {[#PROCS][#HST]} S
    $parray {[[#HST_1_0][#PROCS][#HST_0]] [#HST_1_1]} T

    cufftHandle planxy; cufftPlan2d(&planxy,N,N,CUFFT_C2C);
    $for HST_0(host,hbuf) {
        $copy HST_1(host) to DEV(dev)
        GPU_SAFE(cufftExecC2C(planxy,dev,dev,CUFFT_FORWARD),"fft XY")
        $copy DEV(dev) to HST_1(hbuf)  }
    cufftDestroy(planxy);
    $copy S(hbuf) to T(host)
    cufftHandle planz; cufftPlan1d(&planz,N,CUFFT_C2C,N);
    $for HST_0(host,hbuf) {
        $copy HST_1(host) to DEVT(dev)
        GPU_SAFE(cufftExecC2C(planz,dev,dev,CUFFT_FORWARD),"fft Z")
        $copy DEVT(dev) to HST_1(hbuf)}
    $copy T(hbuf) to S(host)
    cufftDestroy(planz);
}
$end

$main{
    $for k::PROCS {
        _PA_INIT_GPU(0);
        $malloc HSTS_1(host), HSTS_1(hbuf), DEVS(dev)
        if (host==NULL||hbuf==NULL||dev==NULL) exit(-1);
        if (k==0) {
            $malloc HSTS(data1), HSTS(data2)
            $for HSTS(data1) {
                data1->x=(float)rand()/RAND_MAX-0.5;
                data1->y=(float)rand()/RAND_MAX-0.5;}
            $copy HSTS(data1) to HSTS(data2)}
        $copy HSTS(data1) to SCATTER(host)
        $fft($HSTS_1, $DEVS_1, $PROCS)
        $copy SCATTER(host) to HSTS(data1)
        if (k==0) {
            cufftHandle planxyz; cufftPlan3d(&planxyz,N,N,N,CUFFT_C2C);
            $copy HSTS(data2) to DEVS(dev)
            GPU_SAFE(cufftExecC2C(
                planxyz,dev,dev,CUFFT_FORWARD),"fft XYZ")
            $copy DEVS(dev) to HSTS(data2)
            bool res=CompareL2fe((float*)data1,(float*)data2,
                $size(HSTS)*2,0.001);
            printf("Test: %s\n", (1==res)? "PASS.":"FAIL."); }
    }
}
```

7.3 Benchmarking MIC Memory Bandwidth and Floating-Point Performance

A good way to conduct performance optimization is to measure your implementation against the fastest code that can run on the hardware. This provides a measure for how well you have achieved in your current code. For this purpose, we illustrate here a code that runs on Intel MIC (or Xeon Phi) that performs high-speed memory operations (with read:write=1:1) and between adjacent memory accesses double-precision vectorized floating-point operations. Both are intended to push to the limits of the device's capabilities so that both the memory channel and the floating-point units run at near top speeds.

Listing 47 Demo Program: mictest.pa

```
#include <omp.h>
#include <pthread.h>
#include <stdio.h>
#include <immintrin.h>
#define _USE_MIC
$include "parray.pa"
int main(int argc, char *argv[]){_pa_main(); return 0;}
#define N (1<<20)
```

```
#define ncores 59 // #cores-1
#define nthreads_per_core 4
#define ntests 32
$var nrepeats(48)
$parray {paged double[ncores*nthreads_per_core][[N/8][8]]} PAGED
$parray {micmem double # PAGED} MICMEM
$subprog mictest(T)
    $parray {vmem double[8]} VMEM
    $var grid(1)
    $for k::omp[ncores][nthreads_per_core] {
        double* x0=x+$T_0[k]; double* y0=y+$T_0[k];
    #ifdef __MIC__
        $create VMEM(a,b,c)
        for(int i=0; i<8 ; i++) {a[i]=1; b[i]=i; c[i]=k+1;}
        for (int i=0; i<ntests; i++) {
            $create VMEM(z)
            for (int n=0; n<N; n+=8) {
                $copy T_1_1(y0+n) to VMEM(z)
                $repeat(i,0,$nrepeats){
                    for (int j=0; j<8; j++) c[j]=c[j]*a[j]+b[j];}
                $copy VMEM(z) to T_1_1(x0+n)}}
    #endif
    }
$end
$main{
    $create PAGED(x,y)
    $malloc MICMEM(x,y)
    $copy PAGED(x),PAGED(y) to MICMEM(x),MICMEM(y)
    $for mic[1] as(double* x, double* y){
        float Gflop = 1.0e-09*ntests*$size(MICMEM)*$nrepeats*2;
        float Gb = 1.0E-09*ntests*sizeof(double)*$size(MICMEM)*2;
        _PA_CREATE_TIMER
        $mictest($MICMEM) // warmup
        _PA_TIME_NOW
        $mictest($MICMEM)
        _PA_TIME_NOW
        float t = _PA_TIME_DIFF(1,2);
        printf("Bandwidth:\t%.2f GB/s\nPerformance:\t%.2f Gflops\n",
            Gb/t, Gflop/t);
    }
    $copy MICMEM(x) to PAGED(x)
    $destroy PAGED(x,y)
    $destroy MICMEM(x,y)
}
```

If the preprocessor variable "`grid`" is 1 prior to the use of a `mic` array type, then the type is expected to be two-dimensional with the column dimension indicating processor cores and the row dimension indicating threads on each core.

This code pins the threads to the cores available on the MIC processor with affinity set to the number of cores (as the column dimension) and the number of threads per core (as the row dimension) in "`$for omp[ncores][ nthreads_per_core]`". The "`$repeat`" command has been explained in Section 6.7. "`map`" is a pre-defined sub-program in "`parray.pa`" that applies a vector operand to "`vmem`" arrays in the vector registers.

Listing 48 Result(MIC): mictest

```
Bandwidth:      124.53 GB/s
Performance:    747.16 Gflops
```

8 Compiler Environment

PARRAY is designed to work with a variety of compiler environments and hardware devices. Currently the typical Linux and Windows environments are supported. Necessary APIs for different parallelism must be installed, if the source code involves such forms of parallelism. For example, if the source code only uses CPU multi-threading parallelism, only the Pthread API is needed.

8.1 Command Line of Preprocessing Compiler (`pac`)

On Linux, the compilation of a PARRAY program typically consists of several steps. The command line for 64-bit PARRAY preprocessing has syntax:

`pac64` *source_file* *generated_file* <−I *library_include_dir*>

where "`pac`" is the preprocessing command, "`source_file`" is the name of the source file normally with "`.pa`" extension, and "*generated_file*" is the file of the generated code ("`.cu`" for CUDA code and "`.cpp`" for general C++ code), and" *library_include_dir*" indicates the directory location of library files for the "`$include`" command. PARRAY libraries have extension "`.pa`".

The second step is CUDA compilation using "`nvcc`". The generated code should include CUDA, Pthread and MPI header files. The option "`-architecture`" must be appropriately set to generate correct code for the target GPU device. If the system does not have CUDA installed, a normal C compiler will be used. Compilation errors of the generated code (including C errors) also contain the line numbers of their locations in the original source file.

In the next step object files are linked with libraries such as "`cufft`", "`cudart`", "`cublas`", "`cutil`" etc. as well as MPI libraries. In the final step, if MPI is present, the generated binary is launched by "`mpirun`" on multiple processes, which are either allocated to the same number of computing nodes or a number of CPU cores (depending on system options) on a small number of computing nodes.

9 Acknowledgement

PARRAY is a software tool developed under the support of several research grants.