Chapter 5
PARRAY: A Unifying Array Representation for Heterogeneous Parallelism

Yifeng Chen, Xiang Cui and Hong Mei

Abstract This paper introduces a programming interface called PARRAY (or Parallelizing ARRAYs) that supports system-level succinct programming for heterogeneous parallel systems like GPU clusters. The current practice of software development requires combining several low-level libraries like Pthread, OpenMP, CUDA and MPI. Achieving productivity and portability is hard with different numbers and models of GPUs. PARRAY extends mainstream C programming with novel array types of the following features: (1) the dimensions of an array type are nested in a tree structure, conceptually reflecting the memory hierarchy; (2) the definition of an array type may contain references to other array types, allowing sophisticated array types to be created for parallelization; (3) threads also form arrays that allow programming in a Single-Program-Multiple-Codeblock (SPMC) style to unify various sophisticated communication patterns. This leads to shorter, more portable and maintainable parallel codes, while the programmer still has control over performance-related features necessary for deep manual optimization. Although the source-to-source code generator only faithfully generates low-level library calls according to the type information, higher-level programming and automatic performance optimization are still possible through building libraries of sub-programs on top of PARRAY. The case

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Y. Chen (✉) · X. Cui · H. Mei
HCST Key Lab School of EECS, Peking University, Beijing 100871,
People's Republic of China
e-mail: cyf@pku.edu.cn

X. Cui
e-mail: cuixiang08@sei.pku.edu.cn

H. Mei
e-mail: meih@pku.edu.cn

study on cluster FFT illustrates a simple 30-line code that $2\times$-outperforms Intel Cluster MKL on the Tianhe-1A system with 7168 Fermi GPUs and 14336 CPUs.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming · D.3.3 [Programming Languages]: Language Constructs and Features—Concurrent programming structures · D.3.4 [Programming Languages]: Processors—Code generation

General Terms Languages · Performance · Theory

Keywords Parallel programming · Array representation · Heterogeneous paralelism · GPU clusters

5.1 Introduction

Driven by the demand for higher performance and lower hardware and energy costs, emerging supercomputers are becoming more and more heterogeneous and massively parallel. Several GPU-accelerated systems are now ranked among the top 20 fastest supercomputers. Despite the rapid progress in hardware, programming for optimized performance is hard.

The existing programming models are designed for specific forms of parallelism: Pthread and OpenMP for multicore parallelism, CUDA and OpenCL for manycore parallelism, and MPI for clustering. A simple combination of these low-level interfaces does not provide enough support for software productivity and portability across different GPU clusters with varied numbers/models of GPUs on each node. A current common practice is to combine MPI and CUDA. However on a cluster of GPU-accelerated multicore nodes, the number of MPI processes cannot be the number of CPU cores (to use the CPU cores) and the number of GPUs (to use GPUs) at the same time. A seemingly obvious solution is to use the number of GPUs for MPI processes and use OpenMP to control CPU cores, but the complexity of programming with all MPI, CUDA and OpenMP will discourage most application developers.

Such difficulties have led to a variety of new ideas on programming languages (more detailed comparisons in Sect. 5.5). Language design is a tradeoff between abstraction and performance. For high-performance applications, the concern of performance is paramount. We hence ask ourselves a question:

how abstractly can we program heterogeneous parallel systems without introducing noticeable compromises in performance?

The design of PARRAY follows the approach of bottom-up abstraction. That means 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. Our purpose is not to solve all the programmability issues but to provide a bottom level of abstraction on which performance-minded programmers can directly write succinct high-performance code and on which higher-level language features can be implemented without unnecessary performance overheads. This kind of "performance transparency" allows other software layers to be built on top of this layer, and the implementation will not be performance-wise penalized because of choosing PARRAY instead of using low-level libraries directly. A programmer can then choose the right programming level to work with.

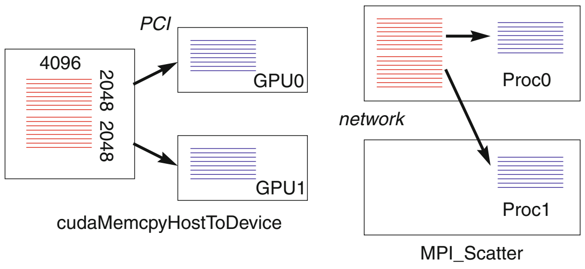
A common means to achieve abstraction is to adopt a unifying communication mechanism, e.g. synchronous message passing in process algebra (Hoare 1985), distributed memory sharing or Partitioned Global Address Space (PGAS). However, heterogeneous parallel systems are often equipped with different kinds of hardware-based communication mechanisms such as sequentially-consistent shared memory for multicore parallelism, asynchronous message-passing or RDMA for clustering, inconsistent shared memory (with explicit but expensive consistency-enforcing synchronization) for manycore parallelism, as well as PCI data transfer between servers and their accelerators. These mechanisms exhibit significantly different bandwidth, latency and optimal communication granularity (i.e. the size of contiguous data segments).

Encoding all data-transfer operations of a source program into a unified communication pattern does not always yield efficient code. For example, programs with assumption of a global-address space (either Distributed Shared Memory or PGAS) tend to issue data-transfer commands when data are needed for computation, but a better strategy may instead use prefetching to better overlap communication and computation. Another challenge is to maximize communication granularity. Shared-memory programs tend to issue individual data-access commands in the actual code for computation, but communication channels such as PCI (between CPU and GPU) and Infiniband (among nodes) require granularity to reach a certain level to achieve peak bandwidths. The source-program information about granularity is easily lost with global-address accesses. Compile-time and runtime optimization alleviates the issues but it remains to be seen to what extent such lost source information is recoverable automatically.

In this paper, we take a different perspective and do not attempt to devise any unifying memory model. Instead we invent novel types that unify the representation of different communication mechanisms. It allows programmers to specify what needs to be done in a communicating operation, and the compiler will then generate the corresponding library calls. This is not restricted to generating hardware-based communication calls. If the required data transfer is one-sided and there exists a well optimized PGAS protocol, the compiler simply generates PGAS calls (Nieplocha et al. 1996) instead.

How can we uniformly represent such a diverse variety of communication mechanisms?

The answer is to identify their common unifying mathematical structure. To get a glimpse of the intended representational unification, consider an example to partition a 4096×4096 row-major array in the main host memory (hmem) to two 2048×4096 blocks, each being copied to a GPU device memory via PCI. Experiments show that the overall bandwidth of two parallel PCI transfers is 1.6x of single PCI transfers. The two CPU multicore threads that control GPUs are therefore programmed to invoke `cudaMemcpy` simultaneously, followed by inter-thread synchronization. Interestingly similar communication patterns arise elsewhere e.g. in collective communication `MPI_Scatter` that scatters the two-block source array to two MPI processes. This is illustrated in the following figures.



Although the two seemingly disparate communication patterns describe data transfers of different library calls (CUDA and MPI), via different media and to different memory devices, they do share the same logical pattern of data partition, distribution and correlation. Such similarities motivate us to unify them as one communication primitive and treat the data layouts of communication's origin and destination as typing parameters.

There have been a large body of research on arrays (e.g. HTA (Ganesh et al. 2006)), but the existing array notations are not expressive and flexible enough to reflect the complex memory hierarchy of heterogeneous parallel systems, control various system-level features, unify sophisticated communication patterns including dimensional transposition, noncontiguous transfer, uncommon internode connections etc., and convey enough source information to the compiler to generate performance-optimal code. The solution is to increase representational expressiveness.

PARRAY adopts a new array type system that allows the programmer to express additional information about parallelization that can guide the compiler to generate low-level library calls. The following features are claimed to be original contributions not seen in other array notations that we know about:

1. Dimension tree
   The dimensions of an array type are nested in a tree, conceptually reflecting the memory hierarchy. Unlike HTA's dimension layers that have default memory types, PARRAY's dimension tree is logical and independent of any specific system architecture. The memory-type information is specifiable for each sub-dimension. Such flexibility allows dimensions to form a hierarchy within individual memory devices.
2. Array-type references
   The definition of an array type may contain references to other array types' definitions. This allows sophisticated array types to be created for transposition, partition, distribution and distortion. Such types form an algebraic system with a complete set of algebraic properties (see Sect. ??).

3. Thread arrays
   Threads also form arrays. A thread array type indicates the kind of processor on which the threads are created, invoked and synchronized. The SPMD codes of different thread arrays are compacted in a nested-loop-like syntactical context (Single-Program Multiple-Codeblocks or SPMC) so that the order of commands in the nested context directly reflects that of the computing task. An array type may consist of a mixture of dimensions that refer to data and thread array types and represent the distribution of array elements over multiple memory devices. A communication pattern is represented as the pairing between such types.

PARRAY essentially organizes various array types in a unifying mathematical framework, but:

how do we know to what extent the basic framework's design is already expressive enough and unnecessary to be extended for future applications and architectural changes?

The answer derives from a theoretical analysis showing that any location-indexing expression (called *offset expression*) for array elements is representable with PARRAY types, as long as it just consists of integer (independent) expressions, multiplication, division and modulo operators, and additions and compositions between expressions. Such level of expressiveness is not seen in any existing array notations.

PARRAY is implemented as a preprocessing compiler that translates directives into C code and macros. As the array types already convey detailed information about the intended communication patterns, the code generation is straightforward and faithful in the sense that the compiler need not second-guess the programmer's intention, and hence little runtime optimization is required. The compiler does generate various conditional-compilation commands, which are usually optimized by the underlying C compiler without causing performance overheads. In a sense, the PARRAY's data-transfer command is like an extremely general MPI collective that invokes the actual communication libraries according to the type parameters. Little overhead-inducing code is generated before or after the invocations. Paradoxically, by not relying on runtime optimization, PARRAY can guarantee performance (for well-coded programs) and form a performance-transparent portable layer of abstraction.

Section 5.2 introduces PARRAY notation. Section ?? studies the mathematical foundation of PARRAY; Sect. 5.3 explains the rationale behind the implementation including the concept of SPMC; Sect. 5.4 investigates into a case study on large-scale FFT; Sect. 5.5 reviews previous works.

5.2 Parallelizing Array Types

This section introduces the PARRAY type notation on which the actual programming syntax is based.

5.2.1 Dimension Tree

Let us first look at a simple definition:

$$A \mathrel{\widehat{=}} \texttt{pinned float}[[3][2]][4].$$

The type describes $3 \times 2 \times 4 = 24$ floating-point elements in three dimensions, but it is also a two-dimensional array type with 6×4 floats and or a one-dimensional array type with 24 floats. The indicated memory type is "`pinned`" which denotes the main host memory allocated by CUDA for fast DMA with GPU.

Note that in the actual code generation, the size of a dimension can be any arbitrary integer expression of C.

As slightly simplified code generation, the compiler translates the *offset expression* (i.e. mapping from indices to element offsets) $A[[x][y]][z]$ into a C expression $x * 8 + y * 4 + z$, $A[x][y]$ into expression $x * 4 + y$, and $A[x]$ into the index x itself. Such offset expressions are used for row-major element accesses in program.

Let the *partial array type* A^0 denote the left subtree of A, i.e. a 2D type of size 3×2, and A^1 denote the right 1D subtree of size 4. The compiler will translate $A^0[x][y]$ into $x * 8 + y * 4$ (instead of $x * 2 + y$)!

5.2.2 Type Reference

A dimtree may contain references to other array types. The following device-memory (or dmem) array type

$$B \mathrel{\widehat{=}} \texttt{dmem float}[[\#A^{01}][\#A^{00}]][\#A^{1}]$$

contains references to A with the sub-dimensions of the left subtree transposed. Their offset expressions satisfy an equation:

$$B[[x][y]][z] = A[[y][x]][z].$$

Unlike the elements of A, those of type B are not contiguously laid out in memory: neither row-major as C convention nor column-major as Fortran. Many sophisticated array layout patterns correspond to the combined uses of dimtree and type references.

5.2.3 Data Transfer

To represent data transfer from an array of type A to an array of type B, we use the following notation:

$$B \leftleftarrows A.$$

The data transfer involves two real array objects in an actual program and describes the operation to copy every element of a type-A array at location $A[i]$ to an element of a type-B array at $B[i]$. In practice, as the source array is located in hmem and the target array in dmem, the compiler will generate:

`cudaMemcpy(..., cudaMemcpyHostToDevice)`

according to the memory types. Other low-level data transfers such as `memcpy` of C, message passing `MPI_Send/MPI_Recv`, collectives `MPI_Alltoall` and so on are generated if the memory types and the dimensions conform to other pairing patterns.

As the offset expressions $A[i]$ and $B[i]$ are not equal, the copying cannot be achieved in a single `cudaMemcpy`. A loop is hence needed to cudaMemcpy every 4 contiguous elements from the starting location $A^0[i]$ in the host memory to the starting location $B^0[i]$ in GPU's device memory as a partially contiguous transfer.

Whether the compiler generates one Memcpy command or a loop of Memcpys depends on the array type's contiguity, which the compiler will generate extra code to check.

5.2.4 Threads Arrays

The following type describes two (Pthread) CPU threads:

$$P \mathrel{\widehat{=}} \texttt{pthd}[2].$$

Here `pthd` is called a *thread type*. No element type is required. Other possible thread types include `mpi` for MPI processes and `cuda` for GPU threads. For example, the type

$$M \mathrel{\widehat{=}} \texttt{mpi}[2]$$

describes two MPI processes. A GPU thread array type is two-dimensional: the column dimension indicates the blocks in a grid, while the row dimension indicates the threads in every block. For example, the following array type

$$C \mathrel{\widehat{=}} \texttt{cuda}[N/256][256]$$

describes N GPU threads of which every 256 threads form a CUDA block. Here N can be any C expression (possibly including runtime variables). The compiler will generate code that lexically includes the size expressions. CUDA thread-array dimensions may contain sub-dimensions just like CUDA's grids and blocks.

5.2.5 Distributed Array Types

If an array type has memory or thread type, its references to other array types only affect the offset expressions, but if that is absent, it becomes *mixed* or *distributed* and are often useful in collective data transfers. Consider an array type in ordinary `paged` memory:

$$H \;\widehat{=}\; \texttt{paged float}[2][[2048][4096]]$$

and another type half of its size in device memory $D \;\widehat{=}\; \texttt{dmem float2}\,\#H^1$. The mixed type

$$[\#P][\#D]$$

logically has the same number of elements as H but does not describe array stored in a single memory device. Instead it is distributed over the threads of P and stored in the GPU device memory associated with each CPU control thread.

5.2.6 PCI Scattering

The communication pattern characterized by the type pairing

$$[\#P][\#D] \;\leftleftarrows\; H$$

describes data copying from every element at location $H[i][j]$ in ordinary paged hmem to the element at location $D[j]$ in the dmem of the GPU that is associated with the control thread $P[i]$. The above communication pattern uses two parallel CPU threads to scatter hmem data to the dmem of two GPUs.

5.2.7 MPI Scattering

The following communication pattern, on the other hand, collectively scatters the source data to two MPI processes:

$$[\#M][\#H^1] \;\leftleftarrows\; H.$$

In practice, a communication pattern may generate different code for synchronous, asynchronous, or one-sided communications.

The communication's mode is determined by the programmer. If that is one-sided, the compiler can generate PGAS code and take advantage of any available runtime optimization (Nieplocha et al. 1996).

5.2.8 Other MPI Collectives

Other MPI collective communications such as Alltoall and Gather have similar a type representation. For example, the following communication pattern describes MPI's Alltoall collective:

$$[[\#H^0][\#M]][\#H^1] \;\leftleftarrows\; [[\#M][\#H^0]][\#H^1]$$

where each element at location $H[j][k]$ on process $M[i]$ is copied to the location $H[i][k]$ on process $M[j]$. It effectively swaps the column dimension of H with the distributed "virtual" dimension M. The compiler detects this pattern and generates the `MPI_Alltoall` command based on the fact that the array type H is contiguous and conforms to the semantics of the MPI command.

Unsurprisingly, MPI gathering accords with the converse communication pattern of MPI scattering:

$$H \;\leftleftarrows\; [\#M][\#H^1].$$

5.2.9 Non-MPI Collectives

Some PARRAY communication patterns do not correspond to any single standard MPI collective. Consider 2x2 four MPI processes:

$$M' \;\widehat{=}\; \texttt{mpi}[2][2]$$

The following pairing of types describes two separate groups of MPI processes performing Alltoall within each individual group:

$$[[\#M'^0][\#H^0][\#M'^1]][\#H^1] \;\leftleftarrows\; [[\#M'^0][\#M'^1][\#H^0]][\#H^1].$$

The pid row dimension M'^1 and the data column dimension H^0 are swapped by the communication.

What gives rise to non-MPI collectives includes not only non-standard inter-process connectivity but also discontiguity in process-to-process communication. In the large-scale 3D FFT algorithm (Chen et al. 2010) developed for turbulence simulation, a distributed array of size up to 14336^3 is transposed over the entire Tianhe-1A GPU cluster with 7168 nodes (see Sect. 5.4). The algorithmic optimization requires discontiguous data transfer between processes to adjust the displacements of elements during communication so that main-memory transposition can then be avoided. The data array of complex numbers (`float2`) on every node has the following type:

$$G \;\widehat{=}\; \texttt{pinned float2}\;[2][[7168][2]][14336].$$

The array type for 7168 MPI processes is defined:

$$L \mathrel{\widehat{=}} \texttt{mpi}[7168].$$

The required discontiguous Alltoall communication pattern (with adjusted displacements) corresponds to the following pairing types:

$$[[\#G^1][\#L][\#G^0]][\#G^2] \;\leftleftharpoons\; [[\#L][\#G^0][\#G^1]][\#G^2]. \tag{5.1}$$

The types have two dimensions. Only the row dimensions are contiguous. The dimension L is swapped with G^1 instead of G^0. That means the starting locations of communicated data are different from those of the standard Alltoall. As the communication granularity i.e. the size of G^2 already reaches 14×8 KB, its performance should be very close to that of a standard Alltoall, despite the fact that no such displacement-adjusting collective exists.[1]

The above examples have illustrated the advantage of adopting a unifying representational framework such that the communication library need not keep adding *ad hoc* collectives to suit originally unforeseen communication patterns that arise from new applications and emerging architectures.

Because all necessary information has been represented in the types, the generated code is as efficient as the underlying MPI implementation. It is also possible to skip the MPI layer and directly generate Infiniband's IB/verbs invocation with less overheads.

5.2.10 Detecting Contiguity

How does the compiler know a communication pattern corresponds to MPI's Alltoall collective or any other sub-routines? This is achieved by checking the memory/thread types of the dimensions and generating C boolean expressions that can check whether the concerned dimensions are *contiguous* in the sense that the offsets of such a dimension are linearly ordered in memory and located adjacently to each other. Usually contiguity-checking expressions are determined in compile-time and induce no performance overheads. Details of this question are beyond our agenda.

5.2.11 Syntax

Let $e, e_0, e_1, \cdots$ denote index expressions in C (variables allowed), the symbols $U, U_0, U_1, \cdots$ denote array types, $S, S_0, \cdots$ be multi-dimensional types. Array types have the following simple syntax:

[1] This communication pattern is equivalent to a loop of asynchronous Alltoallv collectives followed by a global synchronization. Such implementation is possible owing to more recent MPI development (Kandalla et al. 2011).

$$S ::= [U_0]\cdots[U_{n-1}]$$
$$U ::= e \mid \mathtt{disp}\ e \mid S \mid U\#U \mid U^s \mid \mathtt{func}(x)\ e.$$

where s is a path sequence of dimension indices to refer a sub-dimension deep in a dimension tree, and $(U^{s_1})^{s_2}$ is considered the same as $U^{s_1 s_2}$. A C expression e describes a 1D array type of size e. The displacement type ($\mathtt{disp}\ e$) describes a dimension of size 1 with offset e. A dimension may have multiple (up to 10 in practice) sub-dimensions. The dimension size of a type reference $U\#V$ coincides with that of U, while V refers to another type that describes the offset expression. A functional offset expression $\mathtt{func}(x)\ e$ describes a dimension of size 1 with a C offset macro. Functional expressions, as user-defined offset mappings, further strengthen expressiveness but will not be considered in our theoretical analysis.

5.2.12 Informal Rules

Every array-access expression can be decomposed into unary offset expressions called *offset functions*. For example:

$$\begin{aligned} A[[x][y]][z] &= A^{00}[x] + A^{01}[y] + A^{1}[z] \\ &= 8(x\ \mathtt{mod}\ 3) + 4(y\ \mathtt{mod}\ 2) + (z\ \mathtt{mod}\ 4). \end{aligned}$$

That means we only need the definition for every offset function. Another such example is $A^0[x] = 4(x\ \mathtt{mod}\ 6)$. The above semantic definition uses modulo operator. *In the actual implementation the programmer is required to ensure a safe range for every index expression, and the modulo operators are not always generated.*

Consider a type $E \mathrel{\widehat{=}} [2][2]A^0$. The array size is 2×2, but the offset function of E follows that of A^0. The offset function of E^0 is derivable top-down from that of A^0:

$$E^0[x] = A^0[2(x\ \mathtt{mod}\ 2)] = 8(x\ \mathtt{mod}\ 2). \tag{5.2}$$

Then consider the previous example B^0. As its sub-dimensions B^{00} and B^{01} refer to A^{01} and A^{00} respectively, the offset of B^0 is computed bottom-up by decomposing the index into separate indices of its sub-dimensions:

$$\begin{aligned} B^0[x] &= A^{01}[x\ \mathtt{div}\ 3] + A^{00}[x] \\ &= 4((x\ \mathtt{div}\ 3)\ \mathtt{mod}\ 2) + 8(x\ \mathtt{mod}\ 3). \end{aligned} \tag{5.3}$$

The above two cases are intuitive and common in practice. The general array representation, however, allows more sophisticated cases where a dimension itself does not refer to other types but both its parent dimension and some sub-dimensions contain type references. The adopted rule is to first compute top-down according to case (5.2), and then bottom-up according to case (5.3).

5.2.13 Displacement and Index Range

Displacement is a type notation not mentioned in the previous section. It is useful to shift an offset function. In practice we often use $(n..m) \;\widehat{=}\; (m-n+1)\,\#\,(\texttt{disp}\; n)$ to represent a dimension with a range of indices.

The case study in Sect. 5.4 partitions the 3D data in hmem into two-dimensional slices and use GPU to compute the FFT for every slice separately. The type for a slice in dmem is characterized as

$$Q \;\widehat{=}\; \texttt{dmem}\;\texttt{float2}\;[N][N],$$

while one of the slices with displacement in hmem is typed as

$$F \;\widehat{=}\; \texttt{pinned}\;\texttt{float2}\;[\texttt{disp}\;i][N][N].$$

Then the type pairing $Q \leftleftharpoons F$ describes contiguous data transfer of size N^2 from hmem to dmem. The starting location of transfer in hmem is $(N^2 * i)$. We may also declare a smaller two-dimensional "window" in the middle of Q:

$$[4..(N-5)\,\#\,Q^0][4..(N-5)\,\#\,Q^1].$$

Such window types are useful in stencil computation. Cyclic displacement is also easily representable.

5.3 Implementation

This section describes how PARRAY is implemented and the rationale behind it.

5.3.1 Data Array Types

PARRAY is implemented as a C preprocessor that generates CUDA, MPI, and Pthread code and a basic library of sub-programs including the general `DataTransfer` command. The preprocessor is detached from the C compiler to maximize cross-platform compatibility and insensitivity to the constant upgrades of hardware and system-level software. That means only directive errors are caught by the preprocessor. C compilation errors are only detectable in the generated code. The following table compares PARRAY notation and the corresponding program syntax.

The following example shows how an array `a` is declared, created, initialized, accessed and freed in the end:

Notation	Program Syntax
A^{01}	`A_0_1`
$\langle A\rangle$	`$dim(A)$`
$A[5][2]$	`$A[5][2]$`
$A \,\widehat{=}$ `pinned float[[3][2]][4]`	`# parray {pinned float[[3][2]][4]} A`
$\leftleftarrows$	`DataTransfer`

```
#parray {pinned float[[3][2]][4]} A
float* a;
#create A(a)
printf("array access: %d\n", a[$A[5][2]$]);
#destroy A(a)
```

It first declares an array type `A` in pinned memory. The command `#create A(a)` then allocates the pinned memory to the pointer. Note that `#parray` only defines an array type. An actual array is a C pointer that allows multiple typing views as long as the array is a pointer of the element type.

Other memory types include `paged` memory that is managed by the operating system only reaching about 60 % the bandwidth of pinned memory for data transfer with GPU device memory (which is denoted by the keyword `dmem`). The keyword `smem` stands for shared memory in GPU, `rmem` for GPU registers (allocated as direct array declaration in GPU kernels), and `mpimem` for MPI-allocated page-lock DMA-able buffer memory. Mellanox and Nvidia's GPU-Direct technology makes `pinned` and `mpimem` interoperable.

The following table lists the actual library calls for memory allocation. Thread types will be explained in Sect. 3.2.

	`#create`	`#destroy`	Library
`paged`	`malloc`	`free`	C/Pthread
`pinned`	`cudaMallocHost`	`cudaFreeHost`	CUDA
`mpimem`	`MPI_Alloc_mem`	`MPI_Free_mem`	MPI
`dmem`	`cudaMalloc`	`cudaFree`	CUDA
`smem`	`__shared__`		CUDA
`rmem`			CUDA

It is recommended that all index expressions and data transfers should use array notation (instead of native C notation) so that when an array type is modified, all corresponding expressions and library calls will be updated automatically by the compiler over the entire program. The following code declares a type `B` in dmem by referring to `A`'s dimensions and transfers data between their arrays:

```
#parray {dmem float[[#A_0_1][#A_0_0]][#A_1]} B
float* b;
#create B(b)
```

```
#insert DataTransfer(b, B, a, A){}
```

The sub-program `DataTransfer` automatically detects that the dimensions `A_1` and `B_1` are contiguous, and cudaMemcpy data from the main memory to GPU device memory in a loop of 6 segments, each with 4 floats.

The following table lists the library calls used between memory types. `hmem` refers to `paged`, `pinned` and `mpimem` all in the main memory but allocated for different purposes. Data transfer from or to GPU's shared memory `smem` or GPU's registers `rmem` is always performed element-by-element within a loop.

from\to	`hmem`	`dmem`	`smem/rmem`
`hmem`	`memcpy`	`cudaMemcpy`	
`dmem`	`cudaMemcpy`	`cudaMemcpy`	C loop
`smem/rmem`		C loop	C loop

5.3.2 Thread Array Types

SPMD is adopted by a wide range of parallel languages. The idea of SPMD programming is that one code is executed on multiple homogeneous parallel threads (or processes), though at any time different threads may be executing different commands of the code.

SPMD alone does not work for heterogeneous parallelism. For example, FFT on a GPU cluster starts multiple MPI process, each of which initiates several threads to control GPUs on that node, and each thread then launches thousands of GPU threads. Another example is GPU-cluster's Linpack code (Fatica et al. 2009), which performs DGEMM on CPU and GPU threads at the same time.

In the existing explicitly parallel languages, the code segments for different processes are declared separately. The interaction between such code segments requires explicit matching synchronization command. The codeblocks of thread arrays, however, are statically nested in the same program context. The control flow may deviate from one thread array whose code is in an outer codeblock to another thread array whose code is in its immediate inner block and return after the execution of the inner block. The compiler will later extract the codeblocks and sequentially stack those from the same thread array to form a separate SPMD code in which matching synchronization and communication commands are automatically inserted. Thus SPMC (or *Single Program Multiple Codeblocks*) is like a compile-time RPC: the control flows travel across different thread arrays. The purpose is to imply the dynamic control flow as much as possible through the static structure of code and helps the programmer "visualize" the interactive pattern of the control flows among different thread arrays. Each thread array is a SPMD unit consisting of multiple homogeneous processes sharing the same code. A SPMC program may consist of multiple *thread arrays*, each as an array of homogenous threads sharing the same codeblock. The codeblocks of different thread arrays are nested in "one single program".

The following example declares and creates an array of 2 CPU threads and then triggers them to run.

```
#parray {pthd[2]} P
#detour P {printf("thread id %d\n", $tid$);}
```

The expression `$tid$` returns the thread id of the current running thread. The following table lists the actual library calls for creating and freeing thread arrays. The code inside a detour can access global variables as it is generated as a global C function or CUDA kernel in the global context.

	`#create`	`#destroy`	Library
`pthd`	`pthread_create`	`pthread_join`	Pthread
`mpi`	`MPI_Comm_split`	`MPI_Intercomm_merge`	MPI
`cuda`			CUDA

The callee codeblock will be extracted by the compiler separately. If in a program there are two detour commands to the same type of thread array, the callee codeblocks as well as the inserted synchronization commands will be *statically* piled up in the generated C function according to their syntactical order in the source program.

5.3.3 Sub-Programming

A sub-program is like a general C macro function in which array types as well as program codes can be passed as arguments, sometimes achieving surprising flexibility. The following simple code, despite its appearance, directly corresponds to a CUDA kernel that performs general copying within the device memory of a GPU.

```
#subprog GPUDataTransfer(t, T, s, S)
   #parray {cuda($elm(T)$* t,$elm(S)$* s)
                 [$dim(S)$/256][256] } C
   #detour C(t,s){ t[$T[$tid$]$]=s[$S[$tid$]$]; }
#end
```

The arguments `T` and `S` are input types assumed to have an equal (possibly variable) size that is a multiple of 256. The expressions `$elm(T)$` and `$elm(S)$` extract the element types from the input types. The CUDA thread array `C` declares as many threads as the elements with every 256 threads forming a block. The overall thread id `$tid$` here combines CUDA block id and thread id. The inputs `t` and `s` are passed as actual arguments into the CUDA kernel. Every thread copies an element of array `s` at the location `$S[$tid$]$` to the location `$T[$tid$]$` of array `t`. A sub-program is invoked by the command `#insert`. For example the following code declares and creates two arrays of type `Q` in dmem and contiguously copies elements of array *s* to array *t*.

```
#parray {dmem float[N][N]} Q
float* s; #create Q(s)
float* t; #create Q(t)
#insert GPUDataTransfer(t,Q,s,Q)){}
```

This sub-program' effect is the same as

`cudaMemcpy(, , , cudaMemcpyDeviceToDevice)`

as both input types `T` and `S` of the sub-program are contiguous.

With different input types, the sub-program `GPUDataTransfer` may perform a different operation with entirely different generated kernel codes. For example, the following added code effectively performs an out-place array transposition with the row and column dimensions swapped.

```
#parray {dmem float[#Q_1][#Q_0]} R
float* u; #create R(u)
#insert GPUDataTransfer(u,R,s,Q)){}
```

Both data copying and transposition share the same PARRAY code. No existing array representation supports this kind of code reuse. PARRAY sub-programs are insensitive to the layout of the input and output arrays as long as the layout information is described correctly in the declared types. In comparison, a typical function SGEMM for matrix multiplication in the basic math library BLAS has a rather clumsy signature:

```
void sgemm(char transa, char transb,
           int m, int n, int k, ......);
```

where `transa` (or `transb`) is either `'n'` indicating the first array to be row-major or `'t'` being column-major. Other memory layouts are not representable with the BLAS signature. PARRAY's sub-programming mechanism is therefore more flexible than C functions. Like C macros, the price paid for such flexibility is to re-compile a sub-program on every insertion and generate code according to the type arguments.

In fact the general `DataTransfer` command is also a sub-program with the same signature as `GPUDataTransfer`. Its implementation is a series of compile-time conditionals that check the type arguments for structure, memory type and dimension contiguity . The conditional branches eventually lead to various specialized sub-programs like `GPUDataTransfer`.

`GPUDataTransfer` itself too is subject to more specialized optimization. For example, for GPU transposition from Q to R with contiguous column `R_0` and discontiguous `R_1`, shared memory can be used to coalesce both read and write. Other well-known optimization techniques from the CUDA SDK can further minimize bank conflict within shared memory and dmem.

5.4 Case Study: Large-Scale FFT

For *small-scale* FFTs whose data are held entirely on a GPU device, their computation benefits from the high device-memory bandwidth (CUFFT Library Version 2000; Akira and Satoshi 2009; Govindaraju et al. 2008; Nukada et al. 2008; Volkov and Kazian 2008). 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. Compared to a single node, a cluster will provide multiplied memory capacity and bandwidth. 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 fact that GPUs can accelerate large-scale FFTs is surprising, as FFT is extremely bandwidth-intensive, but GPUs *do not* increase network bandwidth.

In our previous work (Chen et al. 2010), we proposed a new FFT algorithm called PKUFFT for GPU clusters. The original implementation uses CUDA, Pthread, MPI and even the low-level infiniband library IB/verbs for performance optimization. That implementation is unportable and specific to a 16-node cluster with dual infiniband cards and dual Tesla C1060 GPUs on each node. To port that code to Tianhe-1A, we first rewrite the code in PARRAY and then re-compile it on the target machine, drastically reducing its length (from 400 lines to 30 lines) while preserving the same depth of optimization. 3D PKUFFT has been deployed to support large-scale turbulence simulation.

Two factors have contributed to the acceleration of FFT with GPUs. Firstly dmem like a giant programmable cache is much larger than CPU cache and hence allows larger sub-tasks to be processed in whole and reduces repeated data transfers between memory and processors. Secondly one 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 the algorithm (Chen et al. 2010) 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 discontiguous process-to-process communications (see Sect. 5.2.9).

At source level, porting code from one platform to another platform is straightforward. For simplicity of presentation, the following code is fixed for N-cubic 3D FFTs and requires the GPU-Direct technology (which was not available originally) to use `pinned` memory as a communication buffer. Without GPU-Direct, the main data and mpimem communication buffer cannot share the same addresses. The variable K is the number of MPI processes.

```
#parray {mpi[K]} L
#detour L(){
 #parray {pinned float2 [N/K][N][N]} G
 #parray {pinned float2 [disp i][N][N]} F
 #parray {dmem float2 [N][N]} Q
 #parray {dmem float2 [#Q_1][#Q_0]} R
 #parray {[[#L][#G_0][#G_1]][#G_2]} S
 #parray {[[#G_1][#L][#G_0]][#G_2]} T
 float2* g; #create G(g)
 float2* gbuf; #create G(gbuf)
 float2* q; #create Q(q)
 float2* qbuf; #create Q(qbuf)
 cufftHandle plan2d;
 cufftPlan2d(&plan2d,N,N,CUFFT_C2C);
 for(int i=0; i<N/K; i++) {
  #insert DataTransfer(q,Q, g,F){}
  cufftExecC2C(plan2d,q,q,CUFFT_FORWARD);
  #insert DataTransfer(gbuf,F, q,Q){}
 }
 #insert DataTransfer(g,T, gbuf,S){}
 cufftHandle plan1d;
 cufftPlan1d(&plan1d,N,CUFFT_C2C,N);
 for(int i=0; i<N/K; i++) {
  #insert DataTransfer(q,Q, g,F){}
  #insert DataTransfer(qbuf,R, q,Q){}
  cufftExecC2C(plan1d,qbuf,qbuf,CUFFT_FORWARD);
  #insert DataTransfer(q,Q, qbuf,R){}
  #insert DataTransfer(gbuf,F, q,Q){}
 }
 #insert DataTransfer(g,S, gbuf,T){}
 cufftDestroy(plan2d);
 cufftDestroy(plan1d);
 #destroy G(g) #destroy G(gbuf)
 #destroy Q(q) #destroy Q(qbuf)
}
```

This code consists of a series of data-transfer operations that we already studied in previous sections. The main 3D complex data are stored in array g of type G. Another array gbuf acts as a buffer. The inner dimension G_2 is contiguous; G_1 is the middle dimension; the outer dimension is a combination of thread dimension L and G_0. Each MPI processes in L contains N/K pages of size N*N. In the first step, every page (with middle and inner dimensions) is transferred to the dmem array q for 2D FFT computation (by calling CUDA library) with results transferred back into `qbuf`. The following communication over the entire network is the non-standard discontiguous Alltoall communication pattern (5.1) that we studied in Sect. 5.2.9. The communication effectively swaps the outer and middle dimensions, so that the middle dimension is aggregated on each MPI process. Every 2D page of the middle and inner dimensions is transferred to dmem again. Before performing batched 1D FFT on the new middle dimension, we use GPU transposition (see Sect. 5.3.3) to swap the middle and inner dimensions to make the middle dimension contiguous. The original positions of the data are restored after FFT by GPU transposition and communication.

In Fig. 5.1 FFT code is tested on Tianhe-1A using up to 7168 nodes, each with 24 GB main memory, two 6-core Intel Processors and one Tesla Fermi 448-core GPU. The special customized network has 80 Gb/s bandwidth for each node and a fat tree structure for switching. CUDA version is 3.0; CUFFT version is 3.1. For comparison, Intel Cluster MKL (or CMKL) 10.3.1.048 is used on the same cluster but does not use GPU. CMKL is already highly optimized because of the heavy communication load of very-large FFTs. The tests are performed for 3D FFT of different sizes for single-precision C2C forward (with returning communication that restores the data to their original positions). Double-precision FFTs perform at the half speed of single precision (on Fermi as well as data transfers). Figure 5.2 (tested for best supported array sizes) shows that on a large GPU cluster, the GPU-based algorithm significantly outperforms CMKL which does not use GPU. GPU-accelerated FFT also scales better than CPU-based FFT. Figure 5.2 illustrates the scalability in more details. Note that we do not swap the outer and inner dimensions directly, as that will affect the granularity of network communication.

Some FFT algorithms (Pekurovsky 2009) adopt two-dimensional decomposition. This is no longer necessary using Parray. On a traditional CPU cluster, the number of MPI processes is usually the number of cores so that MPI utilizes multicore parallelism without the need for OpenMP. That leads to a large number of MPI

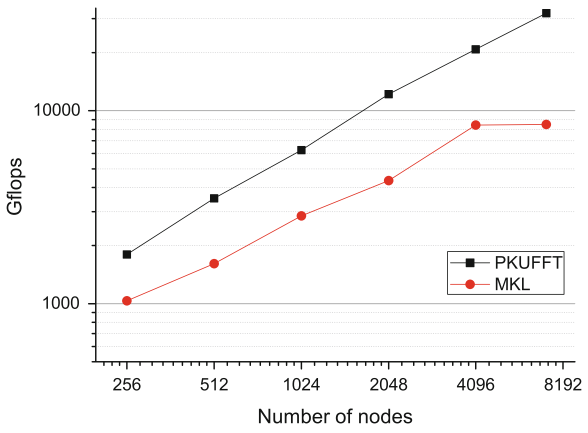


Fig. 5.1 PKUFFT versus intel cluster MKL on Tianhe-1A

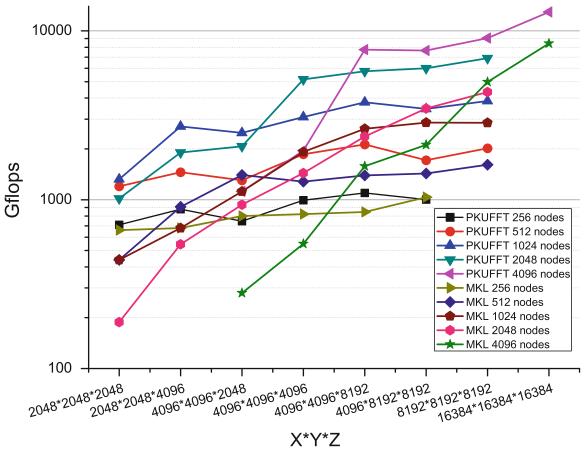


Fig. 5.2 Detailed comparisons of scalability

processes exceeding FFT's dimension size. As we can program each node's internal (multicore and manycore) parallelism, there is no need for over-decomposition that reduces message sizes and performance.

We mainly develop R2C and C2R single-precision 3D FFTs, which are used for large direct numerical simulation of turbulent flows up to the scale of 14336 3D on Tianhe-1A.

5.5 Literature Survey

In this section, we compare our design with other parallel programming interfaces in a table where, for example, "Global" and "Local" denote addressing. The languages considered for comparisons include Chapel (Chamberlain et al. 2007), Co-Array Fortran (CAF) (Numerich and Reid 1998), HMPP (Francois 2010), Hierarchically Tiled Arrays (HTA) (Ganesh et al. 2006), Titanium (Yelick et al. 1998), Stanford PPL (Brown et al. 2011; Chafi et al. 2011), UPC (Zheng et al. 2010), X10 (Charles et al. 2005), ZPL (Chamberlain et al. 2004), Global Arrays (Nieplocha et al. 1996) etc (Fig. 5.3).

These programming interfaces as well as some parallel functional languages (Hains and Mullin 1993) all support some kind of (array) domains. Common dimensional features include Block, Cyclicity, Replication (i.e. indices mapped to replicated value) etc. Such domains are special cases of PARRAY types. CAF also represents multiple cooperating instances of an SPMD program (known as images) through a new type of array dimension called a co-array. Titanium adds several features including multidimensional arrays supporting iterators, subarrays, and copying to Java. ZPL uses a series of array operators to express different access patterns including

	Language	Library	Global	Local	Cluster	Multicore	Manycore	Dimtree/Type Reference
PARRAY	✓			✓	✓	✓	✓	✓
Chapel	✓		✓		✓	✓		
CAF	✓		✓		✓			
HPF	✓		✓		✓			
HMPP	✓			✓		✓	✓	
HTA		✓	✓		✓	✓	✓	
MPI/PVM		✓		✓	✓			
PPL		✓			✓			
Titanium	✓		✓			✓	✓	
UPC	✓		✓		✓			
X10	✓		✓		✓	✓		
ZPL	✓		✓		✓			

Fig. 5.3 Comparisons among parallel programming languages and libraries

translation, broadcast, reduction, parallel prefix operations, and gathers/scatters. In UPC a shared array variable has elements distributed among program instances (or threads). X10 also supports multi-dimensional arrays that can be distributed among cluster nodes.

PARRAY does not directly offer numerical operations between arrays. An operation as simple as matrix multiplication may have different algorithms and implementations on a heterogeneous parallel system. PARRAY is intended to be a performance-transparent layer of abstraction. All effort to lift programming level and hide algorithmic decisions is left to library development of sub-programs.

Among the existing language designs, HTA is perhaps the closest. The claimed points of novelty (1) dimension tree, (2) type references, and (3) thread arrays are not supported by HTA. The improvement is mainly the fact that the array representation in PARRAY is more expressive. The theory part shows a certain algebraic completeness of the representation. Hierarchical tiles arrays assume several default levels (for multicore/cluster parallelism). On the other hand, PARRAY's dimension trees are logical. We believe that hierarchical structure is so important that it should be general and not tied to a specific memory structure.

A large class of languages are PGAS languages. PGAS expects the programmer to think about locality but supports random accesses like shared memory regardless the underlying communication mechanism—essentially something between message passing and variable sharing. PARRAY is not based on any unifying memory model. Instead it is designed to support a variety of communication mechanisms. However, if there exists a well-optimized PGAS library such as Global Arrays, PARRAY can generate code to invoke that library. If the low-level libraries offer both two-sided and one-sided communications, the programmer can use commands `GetDataTransfer` and `PutDataTransfer` to generate one-sided code explicitly.

5.6 Conclusions and Future Work

Our array representation is proposed to unify three forms of parallelism: multicore, manycore and clustering. Other programming ideas often focus on two of them. An advantage of our array representation is its simplicity of semantics and implementation. The source-to-source translation and code generator reach only 2000 lines of C++ code. Basic forms of new data types are intuitive and easy to understand, though some sophisticated types may require more mathematical intuition to handle the inherent complexity of some communication patterns (e.g. the non-standard Alltoall).

Our code generator has been tested on a wide range of other program examples such as matrix operations and stencil computation. In particular we have used FFT in direct numeral simulation of turbulent flows scalable up to 14336^3 single precision. This ongoing experiment (to be reported elsewhere) requires 12 arrays of this size and has reached 36Tflops on Tianhe-1A. Porting the original MPI code to PARRAY for Tianhe-1A took us only five days.

PARRAY only provides abstraction for regular data structures like arrays. Irregular data structures such as trees and graphs must be encoded as arrays to benefit from PARRAY's integrated code generation. The encoding is left to the user or any higher-level software layers/libraries. PARRAY's performance transparency makes sure that any higher-level layers implemented on top of PARRAY will not be performance-wise penalized because of using PARRAY instead of the low-level libraries of its generated code.

The most important thing for a new programming interface is to encourage user acceptance. Training courses have been carried out. Trainees especially those with background in computational sciences respond remarkably well to the new notation. Unlike computer engineers who are more used to language mechanisms like pointer arithmetic, application programmers (e.g. those from oil industry) seem to be more conformable with matrix notation and even find those advanced forms of array types intuitive. For example in stencil computation, a window within a two-dimensional array can be accessed either by moving the pointer to the starting address or using dimensional displacement. Programmers in CS background often prefer pointer arithmetic while many with science backgrounds prefer displacement type. The nature of this interesting difference is perhaps due to their different programming familiarity with C and Fortran, different earlier mathematical education or a combination of the two factors.

In future, PARRAY will support other accelerator devices such as FPGA and Intel MIC and lower-level communication libraries like IB/Verbs. We do not foresee obvious technological hurdles.

Acknowledgments We are grateful to the anonymous referees for their comments that have helped improve the presentation of this paper.

References

Akira N, Satoshi M (2009) Auto-tuning 3D FFT library for cuda GPUs. In: SC'09. ACM, New York, pp 1–10

Brown K et al (2011) A heterogeneous parallel framework for domain-specific languages. In: PACT'11

Chafi H et al (2011) A domain-specific approach to heterogeneous parallelism. In: PPoPP'11

Chamberlain B et al (2004) The high-level parallel language ZPL improves productivity and performance. In: IJHPCA'04

Chamberlain B, Callahan D, Zima HP (2007) Parallel programmability and the Chapel language. IJHPCA 21(3):291–312

Charles P et al (2005) X10: an object-oriented approach to nonuniform cluster computing. In: OOPSLA'05

Chen Y, Cui X, Mei H (2010) Large-scale FFT on GPU clusters. In: ACM International conference on supercomputing (ICS'10), pp 50–59

CUDA CUFFT Library 2009, Version 2.3. NVIDIA Corp.,

Fatica M (2009) Accelerating linpack with CUDA on heterogenous clusters. In: GPGPU'09, June 2009

Francois B (2010) Incremental migration of C and Fortran applications to GPGPU using HMPP. Technical report, hipeac

Ganesh B et al (2006) Programming for parallelism and locality with hierarchically tiled arrays. In: PPoPP'06, pp 48–57

Govindaraju N et al (2008) High performance discrete fourier transforms on graphics processors. In: SC'08, Nov 2008

Hains G, Mullin LMR (1993) Parallel functional programming with arrays. Comput J 36(3):238–245

Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Upper Saddle River

Kandalla K et al (2011) High-performance and scalable non-blocking All-to-All with collective offload on infiniband clusters: a study with parallel 3D FFT. In: ISC'11

Nieplocha JJ, Harrison RJ, Littlefield RJ (1996) Global arrays: a nonuniform memory access programming model for high-performance computers. J Supercomput 10(2):169–189

Nukada A et al (2008) Bandwidth intensive 3-D FFT kernel for GPUs using cuda. In: SC'08, pp 1–11

Numerich R, Reid J (1998) Co-Array Fortran for parallel programming. SIGPLAN Fortran Forum 17(2):1C31

Pekurovsky D (2009) http://www.sdsc.edu/us/resources/p3dfft.php

Volkov V, Kazian B (2008) Fitting FFT onto the G80 architecture. http://www.cs.berkeley.edu/. Accessed May 2008

Yelick K et al (1998) Titanium: a high-performance Java dialect. In: In ACM, pp 10–11

Zheng Y et al Extending unified parallel C for GPU computing. In: SIAM conference on parallel processing for scientific computing