A New Parallel Programming Model for Computer Simulation

Argonne Report ANL/MCS-P5135-0414
Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL, USA
bsmith@mcs.anl.gov.

June 4, 2014

Abstract

The push for ever-increasing levels of concurrency and the increasing complexity of simulation codes require a new parallel programming model that both simplifies the development of extreme-scale codes and makes them more efficient and scalable on newly emerging architectures as well as on GPUs and conventional computing systems. Presented here is a new parallel programming model, inspired by ideas of Dmitry Karpeyev, Matt Knepley, and Jed Brown my co-developers of the Portable Extensible Toolkit for Scientific computation (PETSc), for computer simulation that is simpler than the message-passing-plus-threads parallel computing model, yet still offers high performance while easily supporting telescoping of resources during the computation – a situation that arises, for example, in adaptive mesh refinement codes.

The time has come to replace message passing plus threading as the programming model for scalable computer simulation (also sometimes referred to as high performance computing), yet no good candidates have caught on [4]. This manuscript introduces a new model, called the indexing programming model (IPM), that

- allows for code in a mixture of C, Fortran, C++, CUDA, and OpenCL;
- removes any concept of number of processes or rank of a process from the user interface;
- removes any need for the user to direct the movement of data between processes or memories or the placement of data onto particular processes or memories;
- removes the need for the user to worry about threads or data affinity;
- allows for straightforward and concise adaptive mesh refinement codes;
- provides a programming model that is simpler than MPI [3] plus OpenMP [1];
- is easy to debug with;
- is full featured, allowing complicated sophisticated programs to be written relatively easily;
- provides a resilience model that does not require extensive code from the user, and
- depends on a reasonably sized runtime system that can be written, optimized, and maintained.

In addition, the runtime system manages any resources needed for telescoping of the processor or memory usage as the computation proceeds. For example, as an adaptive mesh refinement runs, if the mesh becomes large enough to require more nodes, they are automatically brought into the computation; and if the mesh becomes smaller, unneeded resources are automatically released. One does not program with a fixed number of processes as is usually done with message-passing codes.

To realize this benefit, of course, the programmer has to give something up; in IPM array access is managed by the runtime, not by pointer arithmetic.

The data in many numerical simulation codes can be thought of as consisting of two parts: numerical values stored in arrays and indices into the numerical value arrays, also stored as arrays, that define the relationship between or meaning of the numerical values (for example, lists of elements in a mesh). Managing the decomposition and movement of the numerical values between nodes in a distributed-memory computer is straightforward because the numerical values remain the same and have the same meaning regardless of where they are located. Doing the same for the indices is difficult, however, because the integer values used as indices have no meaning when they are moved to a different memory without the exact same layout of the numerical arrays. IPM simplifies the management of these indices by automatically remapping the indices to appropriate values when the numerical data and indices are moved between memories. Thus IPM can move the user data freely around the entire distributed-memory machine for data parallelism and load balancing during a simulation without requiring any additional input from the application programmer. IPM provides a small application programmer interface (API) with which the user defines the numerical values (arrays) to be managed and the indices that point into that data. IPM can then move the data around as needed and calls the user's data-parallel task functions on those chunks of data.

One way to think about IPM is that it provides massive task parallelism with a particular API that allows the runtime to efficiently deliver the correct data to the correct task at the correct time. In addition, for many situations it automatically results in data (domain) decomposition parallelism for much of the computation, thus providing excellent data locality. The IPM programming model is intended to scale from modern laptop computers with several processing cores to extreme-scale machines with millions of cores and possibly billions of concurrent operations. However, specific simulations will inevitably be limited in their scalability by the particular algorithms used. To achieve billion-way concurrency still requires appropriate scalable algorithms.

Two important questions about IPM arise:

- **Is the model powerful enough to capture all the needs of numerical simulations?** I believe that the API can handle everything that can be done today with PETSc [2]. I hope it could also be used to code dense matrix operations, sparse direct solvers, multipole methods, and most other numerical schemes.

- **Can one provide efficient implementations of the runtime needed by the model?** The runtime needs to manage partitioning and repartitioning of the data as well as efficient data movement engines and task management. I believe an efficient runtime system can be written on top of MPI and Pthreads or lower-level communication engines.

This document examines in an informal manner how IPM can be used to implement several numerical schemes. The appendix of this document contains a concise description of the IPM API. The sample code can be found at `http://bitbucket.org/IndexingProgrammingModel/ipm`. The file ipm.c contains a crude toy emulator developed to design the IPM API. It uses MPI to manage the distributed-memory emulation and simple sequential stages to emulate multiple threads.

1 Finite Element Method

Consider the representation of a finite element mesh of triangles defined by their vertices. A numerical array would consist of floating-point numbers for each vertex written as

$$V = \{x_0, y_0, x_1, y_1, ..., x_{p-1}, y_{p-1}\},$$

where x_i, y_i are the vertices of point i. A list of triangles could be an array of indices that point into the vertex array

$$E = \{a_0, b_0, c_0, a_1, b_1, c_1, ..., a_{n-1}, b_{n-1}, c_{n-1}\},$$

where a_i, b_i, c_i are the indices that point into V of three vertices of triangle i. Now say one wishes to evaluate a finite element function on this mesh that depends on a single numerical value u_i at each vertex

$$U = \{u_0, u_1, ..., u_{p-1}\}$$

and produces a single numerical value f_i at each vertex

$$F = \{f_0, f_1, ..., f_{p-1}\}.$$

The sequential code could look like the following.

```
F[] = 0
for i=0; i<n; i++
  a,b,c = E[i] // extract the indices of the three vertices
  F[a],F[b],F[c] += ElementFunction(U[a],U[b],U[c],V[a],V[b],V[c])
endfor
```

Note that one can think of "entries" in the E array as 3-tuples and in the V as 2-tuples; that is, $V[i]$ is both the x_i and the y_i numerical values. This approach is used extensively in IPM and in fact goes even further: different "array entries" in IPM arrays may have different sizes. For example, $Q[1]$ might be a single numerical value, whereas $Q[2]$ might have three numerical values. This property is not needed for this simple example.

How could this computation be performed in shared-memory parallelism? Each thread would be assigned a nonoverlap subset of the triangles and would call ElementFunction() for its triangles using the exact same data structures as before. But since different threads would be accumulating values into some of the same locations in $F[]$, the threads each running independently at full speed could result in inconsistent values being placed in those locations. In order to prevent this situation, a ghosted representation of the F[] array is introduced, where vertices that would be accumulated by multiple threads have separate locations in F[], indicated in the array S[]. The resulting code could look like the following.

```
F[] = 0
for i=0; i<n; i++
  a,b,c = E[i]
  ao,bo,co = S[i]
  F[ao],F[bo],F[co] += ElementFunction(U[a],U[b],U[c],V[a],V[b],V[c])
endfor
```

After all the tasks are completed, the values in the ghosted locations in F[] would be accumulated in a nonghosted representation by the runtime system.

How could this be extended to distributed-memory processes with shared-memory threads? One could simply introduce ghosted locations in U[] as well, so each node would have access to all values of U[] needed by the triangles owned by that node. (Here a node refers to all threads that share a common memory.) Again, once all the tasks are completed, the ghosted locations in F[] need to be accumulated into a nonghosted representation.

Sadly, an efficient code for this simple computation in MPI plus OpenMP is a major undertaking. It should not be; it should be trivial. It is not trivial because the user code must perform all the following tasks using MPI calls.

- Partition the triangles among nodes.
- Partition the triangles inside nodes among threads.
- Determine which entries of U[] are needed for each node, and provide message passing to get those values to the correct locations.
- Determine which entries of F[] will be computed for each node.
- Determine which entries of F[] must be ghosted on each memory to prevent contention between different threads accumulating in the same locations.
- Accumulate the partial values for F[] from the ghosted locations for threads and for nodes to compute the unique solution.

The next two subsections introduce the concepts that make these steps straightforward in IPM.

1.1 Creating and Using IPM Arrays

In IPM, data is stored in IPM array objects whose memory is managed by the runtime system. In the main program one creates the abstract arrays to hold the needed quantities. For example,

```
IPM_ArrayNumeric v = IPM_ArrayNumericCreate(&err);
```

is the array that will contain the vertex coordinates[1]. Next one creates the array that will contain the lists of elements.

```
IPM_ArrayIndex e = IPM_ArrayIndexCreate(&err);
```

This is an array whose entries represent indices into other IPM arrays. In this case they will be indices into the `v` array. This is indicated as follows.

```
IPM_ArrayIndexSetOffset(e,IPM_ArrayNumericGetOffset(v,&err),&err);
```

When entries of `e` are partitioned for a computation among nodes and threads, the `v` are also partitioned, and all needed ghost points of `v` will be automatically created and updated.

How does one actually add vertices and triangles to the arrays? This step **cannot** be done from the main program because the main program has no access to the array entries: they are accessible only from *kernel* functions. The following kernel function will put a single triangle and its vertices into the arrays (by convention the name of all kernel functions begin with a k).

```
void kTriCreate(IPMK_ArrayNumeric v,IPMK_ArrayIndex e,IPMK_Error *err)
{
  double     xy[3][2] = {{0,1},{0,0},{1,0}};
  IPMK_Index indices[3];
  indices[0] = IPMK_ArrayNumericAdd(v,2,xy[0],err);
  indices[1] = IPMK_ArrayNumericAdd(v,2,xy[1],err);
  indices[2] = IPMK_ArrayNumericAdd(v,2,xy[2],err);
  IPMK_ArrayIndexAdd(e,3,indices,err);
}
```

This routine adds three entries to the `IPMK_ArrayNumeric` `v`, each consisting of two doubles and one entry to `e` consisting of three `IPMK_Indexes`[2] pointing into `v`. Note that the code saves the indices of the three vertices in the element array but ignores the index output of `IPMK_ArrayIndexAdd()`

[1] For simplicity in this document we consider only a single numerical data type; supporting various precisions are an obvious and easy extension.

[2] In this document a `IPMK_Index` is a 64-bit integer; other choices are possible.

since they will not be needed in this example. In this manuscript all kernel functions are provided in C. If the kernel function is provided in Fortran, however, the runtime system is responsible for presenting the data to the kernel in a way directly accessible by the Fortran compiler. If the kernel is written in CUDA or OpenCL, the runtime system is responsible for ensuring the data is accessible to the GPU (possibly by moving it to the GPU memory) and accessible by the language. Note that within the kernel functions the IPM objects and function names begin with `IPMK_` while in the main program they begin with `IPM_`; this nomenclature emphasizes the fact that although they may refer to the same data, the format inside the kernel may be quite different from that in the main program. For routines and objects that have forms both in the main program and in kernel functions we will use `IPM(K)_` to denote both possible forms.

The following is a kernel that will take arrays of vertices and triangles and refine each triangle into three triangles by adding a vertex in the middle.

```
void kTriRefine(IPMK_ArrayNumeric v,IPMK_ArrayIndex e,IPMK_Error *err)
{
  // loop over my triangles
  for (IPMK_Index p=IPMK_ArrayStart(e,err); p<IPMK_ArrayEnd(e,err); p++){
    const IPMK_Index *ps = IPMK_ArrayIndexGet(e,p,err); // vertices of elements
    const double    *xy0 = IPMK_ArrayNumericGet(v,ps[0],err); // coors of vertex
    const double    *xy1 = IPMK_ArrayNumericGet(v,ps[1],err);
    const double    *xy2 = IPMK_ArrayNumericGet(v,ps[2],err);

    // add new vertex into vertex array
    double     xynew[2] = {(xy0[0]+xy1[0]+xy2[0])/3.0,(xy0[1]+xy1[1]+xy2[1])/3.0};
    IPMK_Index newindex = IPMK_ArrayNumericAdd(v,2,xynew,err);

    // change indices of the old triangle to be one of the three new triangles
    IPMK_Indices indices[3] = {ps[0],ps[1],newindex};
    IPMK_ArrayIndexChange(e,p,3,indices,err);

    // add the two new triangles
    indices[0] = ps[1]; indices[1] = ps[2]; indices[2] = newindex;
    IPMK_ArrayIndexAdd(e,3,indices,err);
    indices[0] = ps[0]; indices[1] = newindex; indices[2] = ps[2];
    IPMK_ArrayIndexAdd(e,3,indices,err);
  }
}
```

Note that the indices into the IPM arrays are abstract entities that have meaning only inside the current instantiation of the kernel function. One can never save the indices (except inside a `IPM_ArrayIndex`) and use them elsewhere. The runtime system manages any transformations of the indices needed between different kernel function calls. Note also there is absolutely no communication between tasks during kernel functions. The runtime system has to ensure that everything needed in a kernel function instance is available when the function begins.

After running the kernel kTriRefine() a number of times, one will have a large number of vertices and triangles distributed across the computational nodes, and the number of nodes used in the computation will increase based on both the problem size and available nodes in the computer system. The runtime system is responsible for partitioning the data across the nodes and threads and calling the kernel functions for each "kernel"'s worth of data.

1.2 Creating and Launching Kernel Functions

IPM has three types of functions:

- Conventional functions that are called directly by the user in C, C++, or Fortran; these can create and destroy IPM arrays but have **no** access to values inside the IPM arrays.
- The same type as above but where the user can launch multiple functions and wait on their completion while also running other functions; these functions handle higher-level task decomposition such as divide and conquer.
- Kernel functions that are managed by the runtime system and have access to values inside appropriate portions of the IPM arrays assigned to them; these functions handle low-level data parallelism.

Note that from the programmer's perspective only a "single instance" of the first two types of functions is running, much like with MPI SIMD usage where in reality a copy of the program is running on each node but conceptually it is often useful to think of it as a single running program. For kernel functions the actual number of instances that are running is irrelevant to the user, much like with "threads" on GPU systems.

Having created some (empty) arrays and written some kernel functions, one needs to call the kernel functions. This is done by first creating a function object and then launching the function with the following.

```
IPM_Function TriCreate = IPM_FunctionCreate(IPM_FUNCTION_REPARTITION,kTriCreate,
                                            2,IPM_MEMORY_WRITE,IPM_MEMORY_WRITE,&err);
IPM_Launch(TriCreate,v,e,&err);
```

The first command tells the runtime system that the kernel function kTriCreate() takes two array arguments, both of which can be changed by the function, and that after the kernel is complete, the runtime system may repartition the resulting arrays. The `IPM_Launch()` command then tells the runtime system to run the kernel function with the given data. Based on the size and current partitioning of the input data, the runtime system will decide how many shared nodes and threads will be used to run the function. Each instance of the kernel function running is a *task*. In this case with TriCreate(), since the arrays are currently empty, only a single thread will be used to run the kernel function, hence a single task. Once the elements have been refined several times, the kernel functions will be run in several tasks. The number of tasks is purely an implementation issue and not something relevant to the application writer.

2 Vector Operations

Vector operations such as $u = a * u + v$ and $d = v' * u$ are fundamental in numerical simulations. These can easily be implemented in IPM. For vector operations often several arrays will have the same number of entries and require access to the same subset of entries in each task. This situation is handled by having these arrays share a common `IPM_Offset`. `IPM_ArrayClone()` results in a new array that shares the previous array's offset.

```
IPM_ArrayNumeric u = IPM_ArrayNumericClone(v,err);
```

Sometimes, the same data may be needed by all instances of the kernel, or values may need to be accumulated across all instances of the kernel, such as in an inner product. These are handled with

```
IPM_RArray A  = IPM_RArrayCreate(1,&err);
```

where one indicates how many entries are in the array, in this case 1. The values can be accessed in the main program or a kernel function with

```
double *a  = IPM(K)_RArrayGet(rvalues,&err);
*a = 25.0;
```

The kernel function for $u = a * u + v$ can be implemented as follows.

```
void kAxpy(IPMK_ArrayNumeric u,IPMK_ArrayNumeric v,IPMK_RArray r,IPMK_Error *err)
{
  double  alpha = *IPMK_RArrayNumericGet(r,err);
  for (IPMK_Index p=IPMK_ArrayStart(u,err); p<IPMK_ArrayEnd(u,err); p++){
    *IPMK_ArrayNumericGet(v,p,err) += alpha*IPMK_ArrayGetDouble(u,p,err);
  }
}
```

Note that the IPM functions used here can be in-lined, resulting in an efficient routine. The `IPM_Function` for this kernel can be created with

```
IPM_Function myAxpy = IPM_FunctionCreate(IPM_FUNCTION_KERNEL,kAxpy,3,
                      IPM_MEMORY_WRITE,IPM_MEMORY_READ,IPM_MEMORY_READ,&err);
```

The kernel for an inner product can be written as

```
void kInner(IPMK_ArrayNumeric u,IPMK_ArrayNumeric v,IPMK_RArray s,IPMK_Error *err)
{
  double     sum = 0;
  for (IPMK_Index p=IPMK_ArrayNumericStart(u,err);p<IPMK_ArrayNumericEnd(u,err);p++){
    sum += *IPMK_ArrayNumericGet(u,p,err)**IPMK_ArrayNumericGet(v,p,err);
  }
  *IPMK_RArrayGet(s,err) = sum;
}
```

and the function object created with

```
  IPM_Function myInner = IPM_FunctionCreate(IPM_FUNCTION_KERNEL,kInner,3,
                         IPM_MEMORY_READ,IPM_MEMORY_READ,IPM_MEMORY_ADD,&err);
```

When a `IPM_RArray` is passed to a kernel function launch, the runtime system is responsible for passing the data to all the tasks. When the `IPM_RArray` is used to accumulate values (indicated by `IPM_MEMORY_ADD` in the function call `IPM_FunctionCreate`), the runtime system is responsible for adding the contributions from all the tasks together. Different tasks never share writable data and hence never need to use locks.

3 Sparse Matrix Vector Products

A sparse matrix vector product produces $v_i = \sum_j A_{ij} u_j$, where almost all the A_{ij} are zero. The most natural way to represent A in IPM is as one numerical `IPM_ArrayNumeric`, A, with one entry per nonzero location and two index `IPM_ArrayIndex`s, Au and Av, to hold the indices for each i and j associated with that numerical value. The kernel function for sparse matrix vector product could then be the following.

```
void kMult(IPMK_ArrayNumeric u, IPMK_ArrayNumeric v,IPMK_ArrayNumeric A,
           IPMK_ArrayIndex Au,IPMK_ArrayIndex Av,IPMK_Error *err)
{
  for (IPMK_Index p=IPMK_ArrayNumericStart(Au,err);p<IPMK_ArrayNumericEnd(Au,err);p++){
    IPMK_Index pu = *IPMK_ArrayIndexGet(Au,p,err);
    IPMK_Index pv = *IPMK_ArrayIndexGet(Av,p,err);
    *IPMK_ArrayNumericGet(v,pv,err) +=
         *IPMK_ArrayNumericGet(A,p,err)*IPMK_ArrayNumericGet(u,pu,err);
  }
}
```

The IPM_Function would be created with

```
IPM_Function myMult = IPM_FunctionCreate(IPM_FUNCTION_KERNEL,kMult,5,
                        IPM_MEMORY_READ,IPM_MEMORY_ADD,IPM_MEMORY_READ,
                        IPM_MEMORY_READ,IPM_MEMORY_READ,&err);
```

Since the second argument is labeled as IPM_MEMORY_ADD, the runtime system is responsible for ensuring that no two tasks will be adding to the same location (based on the locations indicated in Av). The runtime system then accumulates the values at the completion of the tasks.

A drawback to this kernel function is that it may run inefficiently since each nonzero in the sparse matrix results in both a load in u and a store in v. IPM provides kernel contexts that allow tasks to store once-computed information, which can be reused whenever the same task is executed on the same IPMK_Offset data.[3] The call

```
IPMK_OffsetCreatePerKernelContext(IPMK_Offset,size_t,const char name[]);
```

allocates memory that will be available to the task on future calls that see this *exact* offset data with a call to

```
void* IPMK_OffsetGetPerKernelContext(IPMK_Offset,const char name[]);
```

If the offset data is changed as a result of repartitioning or any other reason, the runtime system automatically deletes the context. Thus, to optimize the matrix multiply, the kernel function can store the local part of the matrix in compressed row format in the kernel context and then use that format for the actual local part of the multiply multiple times. The IPM API also provides the ability to create and use contexts associated with offsets and IPM_RArrays; see the appendix.

In certain cases it is necessary to have a IPM_ArrayIndex point into another PMM_ArrayIndex that points into a PMM_ArrayNumeric that holds floating-point numbers. The runtime system is responsible for managing all ghost-point setups required for this case as well.

4 Discussion

Accessing Array Entries: From the examples one can see there are two ways of accessing entries in a IPMK_ArrayNumeric. They may be accessed "contiguously" via an "iterator" or via "indirection" as in

[3] For simplicity we consider names of the context to be simple C/Fortran character strings; more efficient approaches are obviously possible for naming and accessing the contexts.

```
for (IPMK_Index p=IPMK_ArrayNumericStart(Au,err);p<IPMK_ArrayNumericEnd(Au,err);p++){
  IPMK_Index pu = *IPMK_ArrayIndexGet(Au,p,err);
  IPMK_Index pv = *IPMK_ArrayIndexGet(Av,p,err);
  *IPMK_ArrayNumericGet(v,pv,err) +=
       *IPMK_ArrayNumericGet(A,p,err)**IPMK_ArrayNumericGet(u,pu,err);
  }
```

Note that `A` is being accessed via the iterator `p`, while `u` and `v` are accessed via indirection with `pu` and `pv`, which **must** be obtained from `IPM_ArrayIndex`s. Indirection indices can **never** be computed via pointer arithmetic; they can be obtained only via `IPMK_ArrayAddXXX()` and stored in `IPMK_ArrayIndex`s. The only indices that a kernel may provide to `IPMK_ArrayNumericGet()` or `IPMK_ArrayIndexGet()` are those within the ranges of calls to `IPMK_ArrayNumericStart()` and `IPMK_ArrayNumericEnd()` or `IPMK_ArrayNumericGhostStart()` and `IPMK_ArrayNumericGhostEnd()` or are obtained from a `IPMK_ArrayIndexGet()` for a `IPMK_ArrayIndex`, which is associated with the offset of the array using `IPM_ArrayIndexSetOffset()`.

Additionally with the calls

```
  double  *A  = IPMK_ArrayNumericGet(v,pv,err)
  int64_t len = IPMK_ArrayLength(v,pv,err)
```

one can access only values of `A` between `A[0]` and `A[len-1]`; any other values are not defined. That is, one cannot access other entries in `v` by accessing values of `A` outside the legitimate range. This constraint also holds for `IPMK_ArrayIndexGet()`. Note that this gives the runtime system a great deal of freedom in how it actually manages and stores the data, although in a simple implementation values for the next index are likely to be stored adjacent to the values for the previous index.

Resilience and Debugging: Since the runtime system is responsible for the storage locations of all the array data, it can manage resiliency by, for example, storing duplicate copies of the entries across different parts of the machine and hence recover the data from elsewhere if a portion of the machine is lost. Should a failure occur during the running of a kernel function, the runtime system could restart that instance of the kernel function with the original data. This approach won't allow recovery from a failure in the main program; but since essentially all computation takes place in the kernel functions, it could recover from most failures.

For debugging and testing the runtime could serialize its data movement and the launching of the tasks allowing reproducibility (at an enormous performance hit) and the ability to attach a debugger at the launch of any particular task with a known previous state. This approach would eliminate one of the frustrating aspects of debugging message passing and conventional threading code where each time one attaches a debugger (to debug the same problem, with the exact same runtime parameters), one may have a different state in the debugger.

Granularity of Indexing: In the finite element and sparse matrix computations presented here the granularity of the data indicated by a `IPMK_Index` may be small: a single or a handful of double precision numbers. For dense matrix computations in the style of LAPACK, a single index may refer to a substantially large block of say 50×50 doubles. Thus one will see that the IPM model, just like any other programming model, will deliver higher floating-point rates for dense matrix computations than for sparse since the relative cost of the indirection is much lower.

Optimization and Vectorization: The programming model provides a great deal of freedom in the form of the kernel functions. If these are suboptimal, then clearly the performance of the entire code will suffer. Thus attention to performance of the kernels is crucial, and they must be optimized with respect to memory motion as well as vectorization and all the other standard issues that come up in providing efficient sequential code.

Libraries: MPI was carefully designed to allow and support an ecology of libraries. Hence many MPI application codes can avoid having much direct use of MPI and users can avoid having to reinvent the wheel for many common algorithms. We envision a similar model of software libraries around IPM that would require little direct use of IPM by end users.

Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. I thank Dmitry Karpeyev, Matt Knepley, [5] and Jed Brown for their contributions to PETSc that inspired this new unifying model for computer simulation.

Appendix: The API of IPM

```
IPM(K)_ArrayNumeric is an "array" of variable sized numeric items indexed by a IPMK_Index, values in the array are only available in the kernel functions
IPM(K)_ArrayIndex is an "array" of indices indexed by a IPMK_Index, values in the array are available only in the kernel functions

  IPM_ArrayNumeric IPM_ArrayNumericCreate(IPM_Error*)
                   IPM_ArrayNumericClone(IPM_ArrayNumeric,IPM_Error*)
  void             IPM_ArrayNumericCopy(IPM_ArrayNumeric,IPM_ArrayNumeric,IPM_Error*)

  IPM_ArrayIndex   IPM_ArrayIndexCreate(IPM_Error*)
                   IPM_ArrayIndexClone(IPM_ArrayIndex,IPM_Error*)
  void             IPM_ArrayIndexCopy(IPM_ArrayIndex,IPM_ArrayIndex,IPM_Error*)

  double*          IPMK_ArrayNumericGet(IPMK_ArrayNumeric,IPMK_Index index,IPMK_Error*) access values in the array at a particular index
  IPMK_Index*       IPMK_ArrayIndexGet(IPMK_ArrayIndex,IPMK_Index index,IPMK_Error*)
  int64_t          IPMK_ArrayNumericGetLength(IPMK_ArrayNumeric,IPMK_Index,IPMK_Error*)        how many values are at that index
  int64_t          IPMK_ArrayIndexGetLength(IPMK_ArrayIndex,IPMK_Index,IPMK_Error*)

  IPMK_Index IPMK_ArrayNumericStart(IPMK_ArrayNumeric,IPMK_Error*)                    first index of locally owned values
             IPMK_ArrayNumericEnd(IPMK_ArrayNumeric,IPMK_Error*)                     one more than last index of locally owned values
             IPMK_ArrayNumericGhostStart(IPMK_ArrayNumeric,IPMK_Error*)              first index of locally owned ghost values
             IPMK_ArrayNumericGhostEnd(IPMK_ArrayNumeric,IPMK_Error*)                one more than last index of locally owned ghost values

  IPMK_Index IPMK_ArrayIndexStart(IPMK_ArrayIndex,IPMK_Error*)                    first index of locally owned values
             IPMK_ArrayIndexEnd(IPMK_ArrayIndex,IPMK_Error*)                     one more than last index of locally owned values
             IPMK_ArrayIndexGhostStart(IPMK_ArrayIndex,IPMK_Error*)              first index of locally owned ghost values
             IPMK_ArrayIndexGhostEnd(IPMK_ArrayIndex,IPMK_Error*)                one more than last index of locally owned ghost values

  IPMK_Index IPMK_ArrayNumericAdd(IPMK_ArrayNumeric,int64_t,double*,IPMK_Error*)         add new entry to the array; returns index to new location created
             IPMK_ArrayIndexAdd(IPMK_ArrayIndex,int64_t,IPMK_Index*,IPMK_Error*)
  void       IPMK_ArrayNumericChange(IPMK_ArrayNumeric,IPMK_Index,int64_t,double*,IPMK_Error*)    change values in a IPMK_Array
             IPMK_ArrayIndexChange(IPMK_ArrayIndex,IPMK_Index,int64_t,IPMK_Index*,IPMK_Error*)

IPM(K)_Offset provides the parallel/task "layout" of one or more IPM arrays. For tasks to operate on several IPM arrays they generally have the same IPM(K)_Offset

  IPM(K)_OffSet IPM(K)_ArrayNumericGetOffset(IPM_ArrayNumeric,IPM_Error*)
  IPM(K)_OffSet IPM(K)_ArrayIndexGetOffset(IPM_ArrayIndex,IPM_Error*)
  void          IPM_ArrayIndexSetOffset(IPM_ArrayIndex,IPM_Offset,IPM_Error*)    the indices are with respect to the layout defined in supplied offset
                IPM_OffsetCreateContext(IPM_Offset,size_t,const char name[]);     create memory that will be visible in all functions that use the offset
  void*         IPM(K)_OffsetGetContext(IPM_Offset,const char name[]);             this memory is read only in the kernels
  void          IPMK_OffsetCreatePerKernelContext(IPMK_Offset,size_t,const char name[]);  create memory that will be visible to kernel functions with
                                                                                           EXACTLY this same offset data
  void*         IPMK_OffsetGetPerKernelContext(IPMK_Offset,const char name[]);  as soon as arrays are repartitioned this memory is freed. This is intended as a
                                                                                 place a kernel writer can stash data they only want to compute once that could be reused
                                                                                 when the kernel function is re-entered. In a multithreaded implementation each
                                                                                 thread has its own context

IPM(K)_RArray is a "redundant" array that is available in all running kernel functions, its values are accessible in both the kernels and main program
```

```
IPM_RArray IPM_RArrayCreate(int64_t,IPM_Error*)
void       IPM_RArrayDestroy(IPM_RArray*,IPM_Error*)
double*    IPM(K)_RArrayGet(IPM(K)_RArray,IPM_Error*)
void       IPM_RArrayCreateContext(IPM_RArray,size_t,const char name[]);      create some memory that will be visible in all tasks that use the RArray
void*      IPM(K)_RArrayGetContext(IPM(K)_RArray,const char name[]);

IPM_Function is a function object that can launched (started) in the program to perform a computation. Based on its arguments the runtime system
             determines where and on how many "cores" the function is run on

IPM_Function IPM_FunctionCreate(IPM_FunctionOption,void (*func)(IPM_Array,...,IPM_Error*),int64_t nargs,IPM_Memorytype, ...)

IPM_Memorytype
   IPM_MEMORY_READ   - kernel functions can only read values in the array, not write them
   IPM_MEMORY_WRITE  - kernels can read and write values in the array
   IPM_MEMORY_ADD    - kernels can write values into the array, once the kernel functions are complete the results are summed

IPM_FunctionOption
   IPM_FUNCTION_DEFAULT
   IPM_FUNCTION_KERNEL         - it is a kernel function and has access to the array entries
                                 otherwise it is a subprogram and runs asynchronously, call IPM_LaunchWait() to wait until the subprogram is complete
   IPM_FUNCTION_SEQUENTIAL     - each task that runs the function waits for previous task to complete before starting (only for debugging ASCII output)
   IPM_FUNCTION_REPARTITION    - after the function is complete repartition the data across nodes and threads, usually used when the function
                                 adds or deletes many entries in the arrays

void IPM_Launch(IPM_Function,IPM_Array,....,IPM_Error *err)       run the function on the input arrays
     IPM_LaunchWait(PMM_Function,IPM_Error *err)                 wait until the launch function is finished

     IPM_Array is a special value that is used publicly only  to cast the second argument of IPM_Launch
```

References

[1] *OpenMP Application Program Interface, Version 4.0*, OpenMP Architecture Review Board, 2013.

[2] S. Balay, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[3] M. P. I. Forum, *MPI: A Message-Passing Interface Standard, Version 3.0*, High Performance Computing Center Stuttgart (HLRS), 2012.

[4] W. Gropp and M. Snir, *Programming for exascale computers*, Computing in Science and Engineering, 15 (2013), pp. 27–35.

[5] M. G. Knepley and D. A. Karpeev, *Mesh algorithms for PDE with Sieve I: Mesh distribution*, Scientific Programming, 17 (2009), pp. 215–230. http://arxiv.org/abs/0908.4427.