Concept: The Data Decomposition Pattern

CSInParallel Project

July 18, 2014

CONTENTS

Prologue

This document contains reading material that introduces a classic and ubiquitous pattern used in parallel programs: **data decomposition**. Programs containing this pattern perform computations over elements of data stored in linear data structures (and potentially other types of data structures; but we will stick to linear here). In non-parallel, or 'serial', or 'sequential' implementations of programs that contain linear data structures, we often iterate over all the data elements using a for loop. In parallel implementations, we need to decide which elements will be computed by multiple processing units at the same time. We dub the choices we make in our parallel implementation for achieving this the **data decomposition pattern**, because we will chose a decomposition, or mapping of elements in the data structure to processing units available to us. We introduce an example analogous to "Hello World" for this data decomposition programming pattern: addition of vectors, using the simple linear array data structure. We have code examples for different types of hardware and software that enable parallel computation.

The first two chapters introduce the problem and describe ways to decompose it onto processing units. The next three chapters show examples of just how this is done for three different hardware and software combinations. We wrap up with alternative mapping schemes and some questions for reflection.

Nomenclature

A **Processing Unit** is an element of software that can execute instructions on hardware. On a multicore computer, this would be a *thread* running on a core of the multicore chip. On a cluster of computers, this would be a *process* running on one of the computers. On a co-processor such as a Graphics Processing Unit (GPU), this would be a *thread* running on one of its many cores.

A program that uses only one procesing unit is referred to as a *serial* or *sequential* solution. You are likely most familiar with these types of programs.

Prerequisites

- Knowledge of C programming language is helpful.
- **Basic understanding of three types of parallel and distributed computing (PDC) hardware:**
    - Shared-memory multicore machines
    - Clusters of multiple computers connected via high-speed networking
    - Co-processor devices, such as graphical processing units (GPU)
- **Though not strictly necessary, if you want to compile and run the code examples, you will need unix-based computers wit**
    - MPI installed on a single computer or cluster (either MPICH2 or OpenMPI)
    - gcc with OpenMP (most versions of gcc have this enabled already) and a multicore computer
    - CUDA developer libraries installed on a machine with a CUDA-capable GPU

We have compiled and run these on linux machines or clusters with a few different GPU cards.

Code Examples

You can download `VectorAdd.tgz` to obtain all of the code examples shown in the following sections of this reading.

CHAPTER

ONE

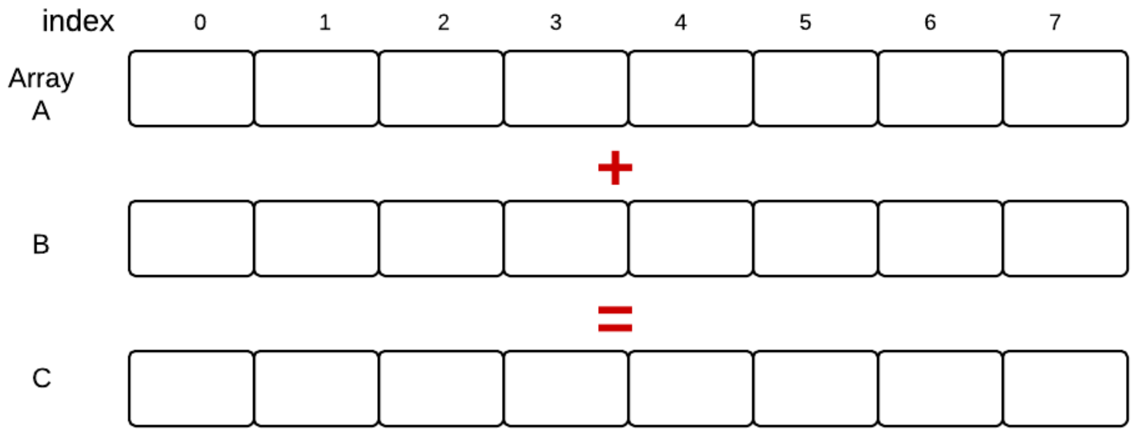
VECTOR ADDITION

The problem we will examine is this: we wish to add each element of vector A with its corresponding element in vector B and placing the sum of the two elements in its corresponding location in vector C. This example problem has sometimes been called the “Hello, World” of parallel programming. The reasons for this are:

- the sequential implementation of the code is easy to understand
- the pattern we employ to split the work is used many times in many situations in parallel programming

Once you understand the concept of splitting the work that we illustrate in this example, you should be able to use this pattern in other situations you encounter.

The problem is quite simple and can be illustrated as follows:

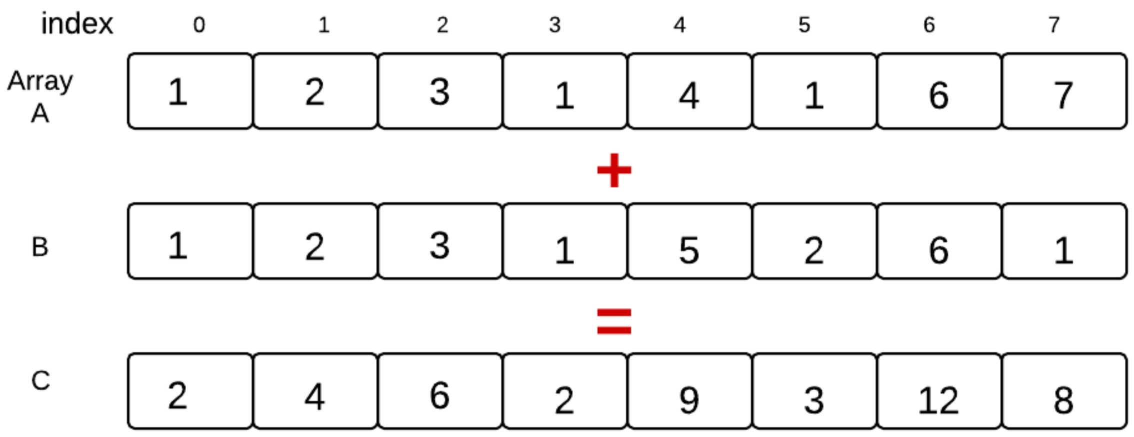


We have two arrays, A and B, and array C will contain the addition of corresponding elements in A and B. In this simple example we are illustrating very small arrays containing 8 elements each. Suppose those elements are integers and A and B had the following elements in it:

The elements in array C above depict the result of adding a vector as stored in array A to a vector as stored in array B.

We use very small vectors of size 8 for illustration purposes. A sequential solution to this problem, written in C code, is found in the file named **VectorAdd/Serial/VA-sequetial.c** in the compressed tar file of examples that accompanies this reading. It looks like this:

```
#include <stdlib.h>   //malloc and free
#include <stdio.h>    //printf
```



```
#define ARRAY_SIZE 8     //Size of arrays whose elements will be added together.

/*
 *  Classic vector addition.
 */
int main (int argc, char *argv[])
{
        // elements of arrays a and b will be added
        // and placed in array c
        int * a;
        int * b;
        int * c;

        int n = ARRAY_SIZE;   // number of array elements
        int i;       // loop index

        // allocate spce for the arrays
        a = (int *) malloc(sizeof(int)*n);
        b = (int *) malloc(sizeof(int)*n);
        c = (int *) malloc(sizeof(int)*n);

        // initialize arrays a and b with consecutive integer values
        // as a simple example
        for(i=0; i<n; i++) {
            a[i] = i;
        }
        for(i=0; i<n; i++) {
            b[i] = i;
        }

        // Compute the vector addition
        for(i=0; i<n; i++) {
                c[i] = a[i]+b[i];
        }

        // Check for correctness (only plausible for small vector size)
        // A test we would eventually leave out
41         printf("i\ta[i]\t+\tb[i]\t=\tc[i]\n");
42         for(i=0; i<n; i++) {
43                 printf("%d\t%d\t\t%d\t\t%d\n", i, a[i], b[i], c[i]);
44         }
45 
46         // clean up memory
47         free(a);  free(b); free(c);
48 
49         return 0;
50 }
```

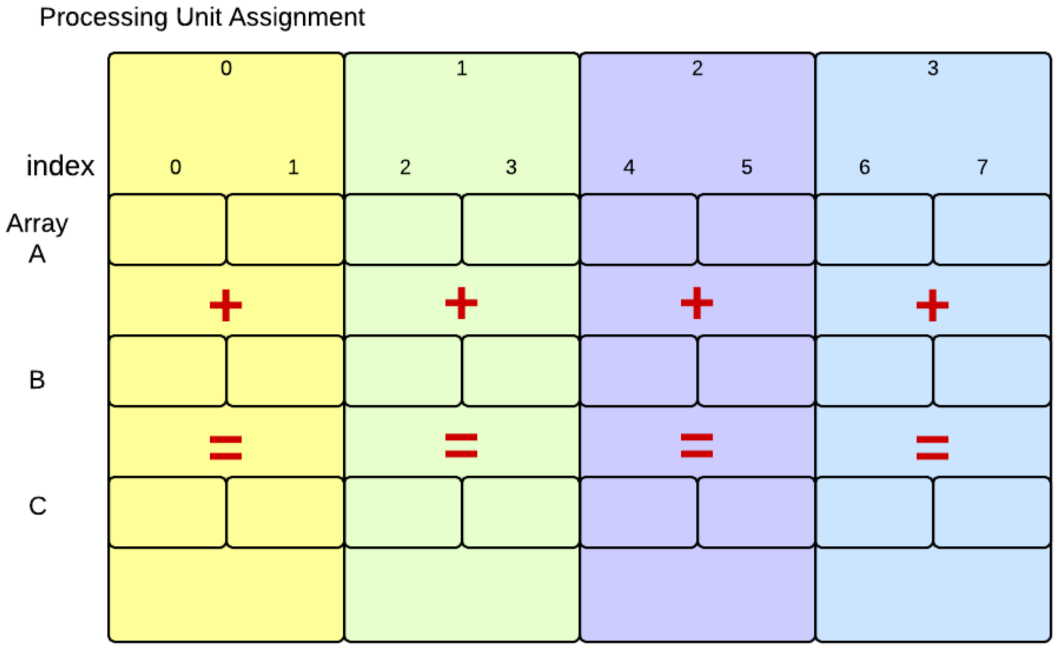
Note the for loop that is doing the actual work we desire, beginning on line 35. This depicts what we sometimes refer to as the ‘do N times’ pattern in classical sequential programming. In the next section we will describe how we consider using multiple processing units to do this work in parallel.

CHAPTER

TWO

DECOMPOSITION OF DATA MANIPULATION TASKS

We have seen how we can complete the task of adding corresponding elements by *sequentially* performing the additions, one at a time, on each element of arrays A and B, placing the result in array C. Now suppose we want to put multiple *processing units* to work on this problem in parallel, concurrently. There are a few different ways to accomplish this. Using the example from the previous section, let's look at one possible way that we might map the computation to processing units:



Here we see our original problem with eight elements in each array. Processing unit 0 would work on the first two elements, processing unit 1 would work on the next two, and so on. Theoretically, using four processing units executing concurrently, the work should get done in one-fourth the time it takes to run it sequentially. In practice, this never quite happens, because of the overhead to start processing units and possible contention for resources, such as the arrays themselves.

For many larger problems with more sophisticated computations, however, this type of decomposition of computational tasks on data elements is very common and can speed up your program. There is no standard terminology for this version of data decomposition; we like to refer to it as *decomposing into equal-sized chunks*. There are other possible types of decomposition and mapping of processing units to elements (perhaps others have been running through your mind). We limit ourselves to this particular way of decomposing, because:

- it occurs quite often,
- it is a good way to do it in many situations because elements are stored contiguously in memory, and
- you can concentrate on and learn one way of using multiple processing units.

Next we will examine example code for three combinations of parallel software and hardware:

- Message passing using the MPI library, which can be used on a cluster or a single multicore computer.
- Shared memory and threads using the OpenMP compiler directives in gcc, which can be used on a multicore computer.
- CUDA programming with the nvcc compiler, which can be used on a machine with a Graphics Procesing Unit, or GPU.

CHAPTER

THREE

VECTOR ADD WITH MPI

Message passing is one way of distributing work to multiple *processes* that run indepentdently and concurrently on either a single computer or a cluster of computers. The processes, which are designated to start up in the software and are run by the operating system of the computer, serve as the processing units. This type of parallel programming has been used for quite some time and the software libraries that make it available follow a standard called Message Passing Interface, or MPI.

One feature of MPI programming is that one single program designates what all the various processes will do– a single program runs on all the processes. Each process has a number or *rank*, and the value of the rank is used in the code to determine what each process will do. In the following code, the process numbered 0 does some additional work that the other processes do not do. This is very common in message passing solutions, and process 0 is often referred to as the master, and the other processes are the workers. In the code below, look for three places where a block of code starts with this line:

```
if (rank == MASTER)  {
```

This is where the master is doing special work that only needs to be done once by one process. In this case, it is the initialization of the arrays at the beginning of the computation, the check to ensure accuracy after the main computation of vector addition is completed, and freeing the memory for the arrays at the end.

The MPI syntax in this code takes some getting used to, but you should see the pattern of how the data decomposition is occuring for a process running this code:

1. First initialize your set of processes (the number of processes in designated when you run the code).
2. Determine how many processes there are working on the problem.
3. Determine which process rank I have.
4. If I am the master, initialze the data arrays.
5. Create smaller arrays for my portion of the work.
6. Scatter the equal-sized chunks of the larger original arrays from the master out to each process to work on.
7. Compute the vector addition on my chunk of data.
8. Gather the completed chunks of my array C and those of each process back onto the larger array on the master.
9. Terminate all the processes.

The following code contains comments with these numbered steps where they occur. This is the file **VectorAdd/MPI/VA-MPI-simple.c** in the compressed tar file of examples that accompanies this reading.

```
/*
 * Prerequisties:
 *     This code runs using an MPI library, either OpenMPI or MPICH2.
 *     These libraries can be installed in either a cluster of computers
 *     or a multicore machine.
 *
 *  How to compile:
 *     mpicc -o vec-add VA-MPI-simple.c
 *
 *  How to execute:
 *     mpirun -np 2 ./vec-add
 *
 *     Note that this executes the code on 2 processes, using the -np command line flag.
 *     See ideas for further exploration of MPI using this code at the end of this file.
 */


#include "mpi.h"      // must have a system with an MPI library
#include <stdio.h>    //printf
#include <stdlib.h>   //malloc

/*
 * Definitions
 */
#define MASTER 0          //One process will take care of initialization
#define ARRAY_SIZE 8      //Size of arrays that will be added together.

/*
 *  In MPI programs, the main function for the program is run on every
 *  process that gets initialized when you start up this code using mpirun.
 */
int main (int argc, char *argv[])
{
        // elements of arrays a and b will be added
        // and placed in array c
        int * a;
        int * b;
        int * c;

        int total_proc;         // total nuber of processes
        int rank;        // rank of each process
        int n_per_proc;        // elements per process
        int n = ARRAY_SIZE;   // number of array elements
        int i;       // loop index

        MPI_Status status;   // not used in this arguably poor example
                             // that is devoid of error checking.

        // 1. Initialization of MPI environment
        MPI_Init (&argc, &argv);
        MPI_Comm_size (MPI_COMM_WORLD, &total_proc);
        // 2. Now you know the total number of processes running in parallel
        MPI_Comm_rank (MPI_COMM_WORLD,&rank);
        // 3. Now you know the rank of the current process

        // Smaller arrays that will be held on each separate process
            int * ap;
        int * bp;
        int * cp;

        // 4. We choose process rank 0 to be the root, or master,
        // which will be used to  initialize the full arrays.
        if (rank == MASTER)  {
                a = (int *) malloc(sizeof(int)*n);
                b = (int *) malloc(sizeof(int)*n);
                c = (int *) malloc(sizeof(int)*n);

                // initialize arrays a and b with consecutive integer values
                // as a simple example
                for(i=0;i<n;i++)
                        a[i] = i;
                for(i=0;i<n;i++)
                        b[i] = i;
        }

        // All processes take part in the calculations concurrently

        // determine how many elements each process will work on
        n_per_proc = n/total_proc;
        /////// NOTE:
        // In this simple version, the number of processes needs to
        // divide evenly into the number of elements in the array
        ///////////

        // 5. Initialize my smaller subsections of the larger array
        ap = (int *) malloc(sizeof(int)*n_per_proc);
        bp = (int *) malloc(sizeof(int)*n_per_proc);
        cp = (int *) malloc(sizeof(int)*n_per_proc);

        // 6.
        //scattering array a from MASTER node out to the other nodes
        MPI_Scatter(a, n_per_proc, MPI_INT, ap, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);
        //scattering array b from MASTER node out to the other node
        MPI_Scatter(b, n_per_proc, MPI_INT, bp, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);

        // 7. Compute the addition of elements in my subsection of the array
        for(i=0;i<n_per_proc;i++)
                cp[i] = ap[i]+bp[i];

        // 8. MASTER node gathering array c from the workers
        MPI_Gather(cp, n_per_proc, MPI_INT, c, n_per_proc, MPI_INT, MASTER, MPI_COMM_WORLD);

/////////////////////// all concurrent processes are finished once they all communicate
/////////////////////// data back to the master via the gather function.

        // Master process gets to here only when it has been able to gather from all processes
        if (rank == MASTER)  {
                // sanity check the result  (a test we would eventually leave out)
                int good = 1;
                for(i=0;i<n;i++) {
                        //printf ("%d ", c[i]);
                        if (c[i] != a[i] + b[i]) {
                                printf("problem at index %lld\n", i);
                                good = 0;
                                break;
                        }
                }
                if (good) {
                        printf ("Values correct!\n");
                }

        }

        // clean up memory
        if (rank == MASTER)  {
                free(a);  free(b); free(c);
        }
        free(ap);  free(bp); free(cp);

        // 9. Terminate MPI Environment and Processes
        MPI_Finalize();

        return 0;
}
```

CHAPTER

FOUR

VECTOR ADD WITH OPENMP

Computers with multicore processors and a single shared memory space are the norm, including not only laptops and desktops, but also most phones and tablets. Using multiple cores concurrently on these machines, can be done in several programming languages; we will demonstrate the use of C with a set of compiler directives and library functions known as OpenMP. The OpenMP standard is built into many C compilers, including gcc on unix machines.

OpenMP on shared memory multicore machines creates *threads* that execute concurrently. The creation of these threads is implicit and built by the compiler when you insert special directives in the C code called *pragmas*. The code that begins executing main() is considered thread 0. At certain points in the code, you can designate that more threads should be used in parallel and exucute concurrently. This is called *forking* threads.

In the code below, you will see this pragma, which is implicitly forking the threads to complete the computation on equal chunks of the orginal array:

```
#pragma omp parallel for shared(a, b, c) private(i) schedule(static, 2)
```

The `shared` keyword indicates that the arrays are shared in the same memory space for all threads, and the `private` keyword indicates that each thread will have its own copy of the index counter i that it will increment.

The `schedule` keyword is used in this pragma to indicate how many consecutive iterations of the loop, and thus computations on consecutive elements of the arrays, that each thread will execute. In data decomposition, we like to call this the **chunk size** assigned to each thread (not necessarily a universal term, but hopefully it conveys the idea). To mimic our simple 8-element example, this code (shown below) sets the number of threads to 4 and the chunk size to 2.

The syntax of this OpenMP code example below is very similar to the original sequential version. In fact, it was derived from the sequential version by adding this pragma, including the OpenMP library, called omp.h, setting how many threads to use and the chuck size just before the forking, and adding some print statements to illustrate the decomposition and verify the results.

This pragma around for loops is built into openMP because this ‘repeat N times” pattern occurs so frequently in a great deal of code. This simplicity can be deceiving, however– this particular example lends itself well to having the threads share data, but other types of problems are not this simple. This type of data decomposition example is sometimes called *embarassingly parallel*, because each thread can read and update data that no other thread should ever touch.

This code is the file **VectorAdd/OpenMP/VA-OMP-simple.c** in the compressed tar file of examples that accompanies this reading.

```
#include <stdlib.h>   //malloc and free
#include <stdio.h>    //printf
#include <omp.h>      //OpenMP

// Very small values for this simple illustrative example
#define ARRAY_SIZE 8     //Size of arrays whose elements will be added together.
#define NUM_THREADS 4    //Number of threads to use for vector addition.
```

```

/*
 *  Classic vector addition using openMP default data decomposition.
 *
 *  Compile using gcc like this:
 *          gcc -o va-omp-simple VA-OMP-simple.c -fopenmp
 *
 *  Execute:
 *          ./va-omp-simple
 */
int main (int argc, char *argv[])
{
        // elements of arrays a and b will be added
        // and placed in array c
        int * a;
        int * b;
        int * c;

        int n = ARRAY_SIZE;                 // number of array elements
        int n_per_thread;                   // elements per thread
        int total_threads = NUM_THREADS;    // number of threads to use
        int i;       // loop index

        // allocate spce for the arrays
        a = (int *) malloc(sizeof(int)*n);
        b = (int *) malloc(sizeof(int)*n);
        c = (int *) malloc(sizeof(int)*n);

        // initialize arrays a and b with consecutive integer values
        // as a simple example
        for(i=0; i<n; i++) {
            a[i] = i;
        }
        for(i=0; i<n; i++) {
            b[i] = i;
        }

        // Additional work to set the number of threads.
        // We hard-code to 4 for illustration purposes only.
        omp_set_num_threads(total_threads);

        // determine how many elements each process will work on
        n_per_thread = n/total_threads;

        // Compute the vector addition
        // Here is where the 4 threads are specifically 'forked' to
        // execute in parallel. This is directed by the pragma and
        // thread forking is compiled into the resulting exacutable.
        // Here we use a 'static schedule' so each thread works on
        // a 2-element chunk of the original 8-element arrays.
        #pragma omp parallel for shared(a, b, c) private(i) schedule(static, n_per_thread)
        for(i=0; i<n; i++) {
                c[i] = a[i]+b[i];
                // Which thread am I? Show who works on what for this samll example
                printf("Thread %d works on element%d\n", omp_get_thread_num(), i);
        }

        // Check for correctness (only plausible for small vector size)
        // A test we would eventually leave out
        printf("i\ta[i]\t+\tb[i]\t=\tc[i]\n");
        for(i=0; i<n; i++) {
                printf("%d\t%d\t\t%d\t\t%d\n", i, a[i], b[i], c[i]);
        }

        // clean up memory
        free(a);  free(b); free(c);

        return 0;
}
```

CHAPTER

FIVE

VECTOR ADD WITH CUDA

Using the CUDA C language for general purpose computing on GPUs is well-suited to the vector addition problem, though there is a small amount of additional information you will need to make the code example clear. On GPU co-processors, there are many more cores available than on traditional multicore CPUs. Each of these cores can execute a *thread* of instructions indicated in the code. Unlike the threads and processes of OpenMP and MPI, CUDA adds extra layers of organization of the threads: programmers set up *blocks* containing a certain number of threads, and organize the blocks into a *grid*.

For the problem we originally posed, where we had 8 elements in an array, one possible organization we could define in CUDA would be 4 blocks of 2 threads each, like this:

CUDA convention has been to depict treads as squiggly lines with arowheads, as shown above. In this case, the 4 blocks of threads that form the 1-dimensional grid become analogous to the processing units that we would assign to the portions of the array that would be run in parallel. In contrast to OpenMP and MPI, however, the individual threads within the blocks would each work on one data element of the array [1].

5.1 Determining the thread number

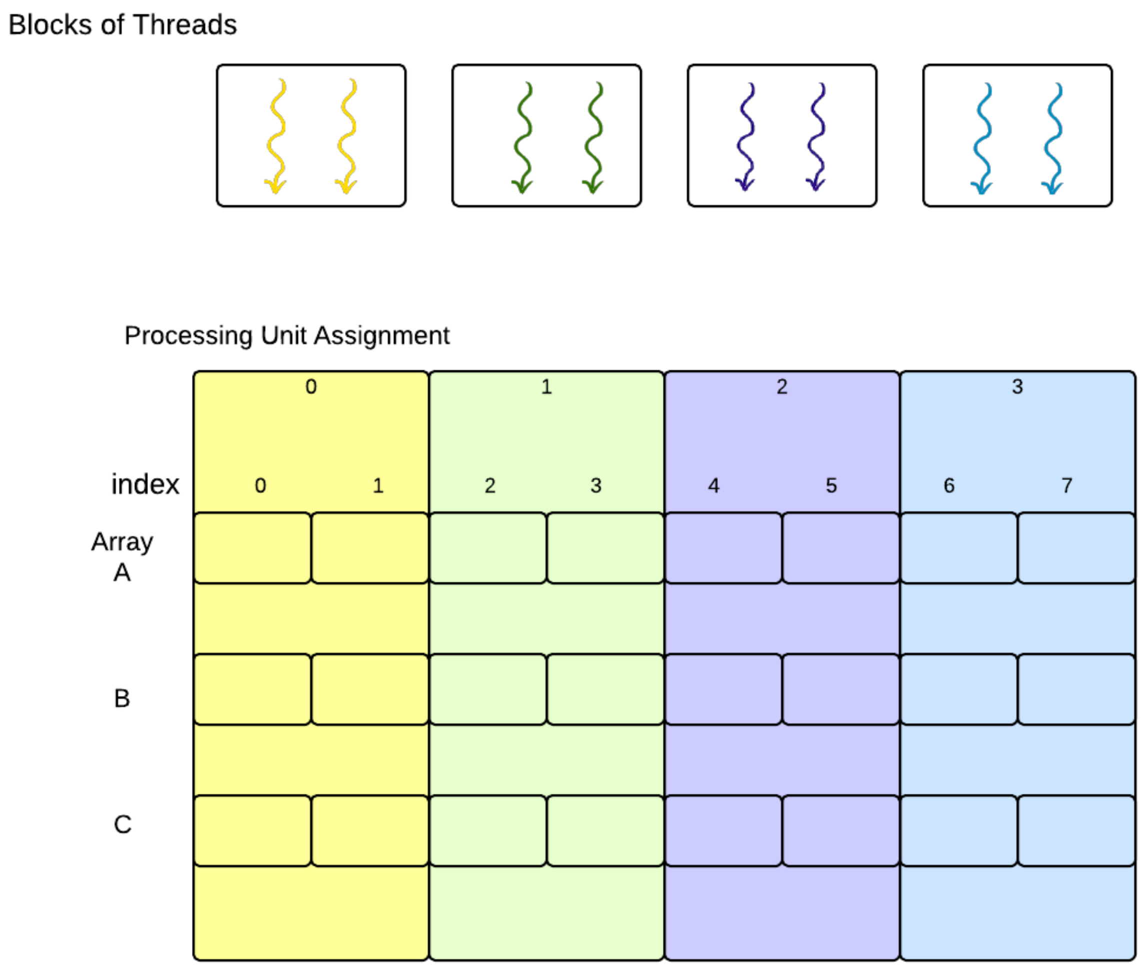
In all parallel programs that define one program that all threads will run, it is important to know which thread we are so that we can calculate what elements of the original array are assigned to us. We saw with the MPI program, we could determine what the 'rank' of the process was that was executing. In the OpenMP example in the previous section, thread number assignment to sections of the array in the for loop was implicit [2]. In CUDA programs, we always determine which thread is running and use that to determine what portion of data to work on. The mapping of work is up to the programmer.

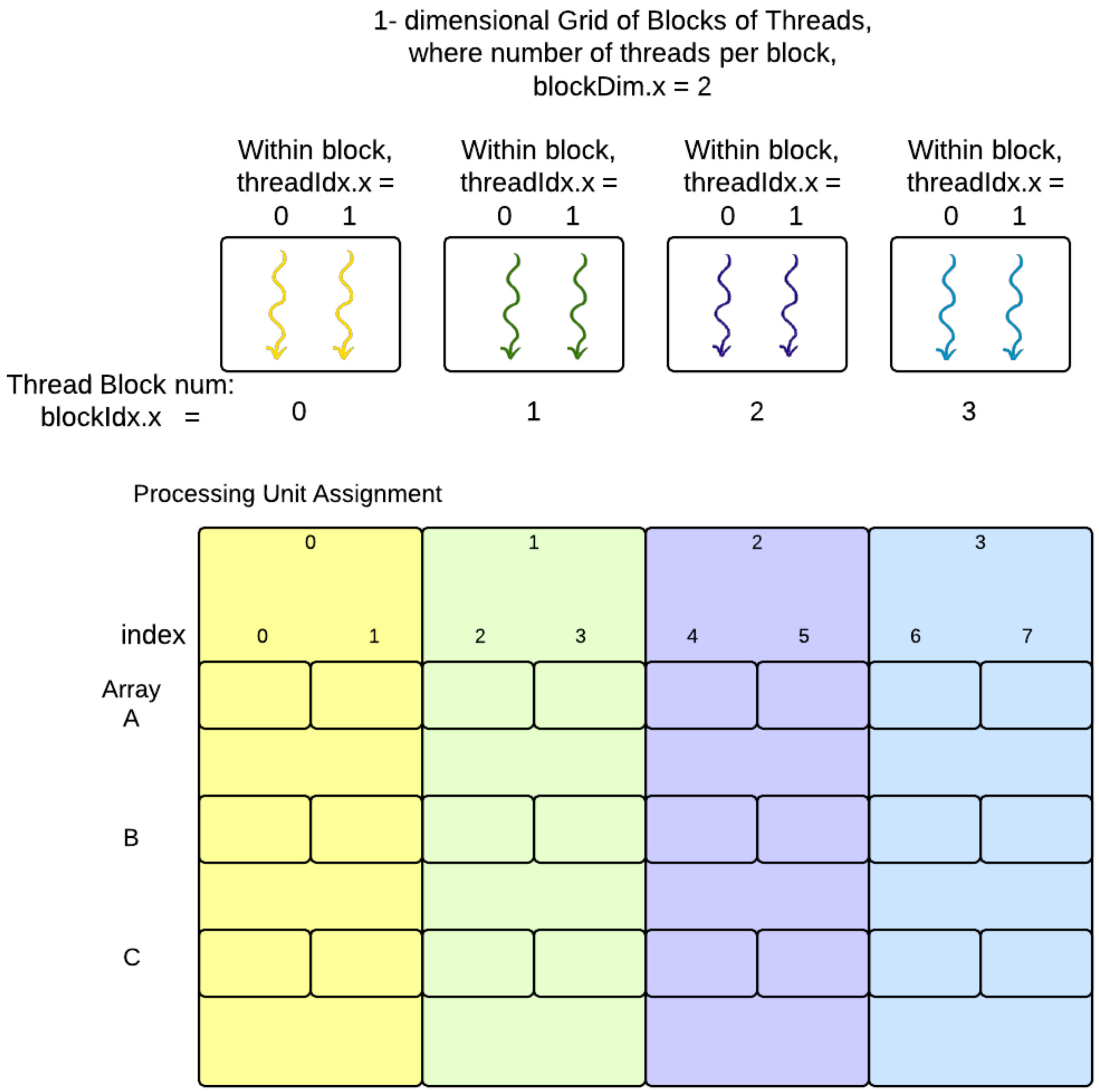
In CUDA programs, we set up the number of blocks that form the grid and we define how many threads will be used in each block. Once these are set, we have access to variables supplied by the CUDA library that help us define just what thread is executing. The following diagram shows how three of the available variables, corresponding to the grid, blocks, and threads within the blocks would be assigned for our above example decomposition:

In this case of 1-dimensional arrays whose elements are being added, it is intuative to use a 1-dimensional grid of blocks, each of which has a 1-dimensional set of threads within it. For more complicated data, CUDA does let us define two or three-dimensional groupings of blocks and threads, but we will concentrate one this 1-dimensional example here. In this case, we will use the 'x' dimension provided by CUDA variables (the 'y' and 'z' dimensions are

[1] Each GPU, depending on what type it is, can run a certain number of CUDA blocks containing some number of threads concurrently. Your code can define the number of blocks and the number of threads per block, and the hardware will run as many as possible concurrently. The maximum number of threads you can declare in a block is 1024, while the number of blocks you can declare is a very large number that depends on your GPU card. Here we show a simple case of four blocks of 2 threads each, which in CUDA terminology forms a grid of blocks. This particular grid of blocks would enable each element in an array of size 8 to be computed concurrently in parallel by 8 CUDA threads. To envision how the decomposition of the work might occur, imagine that the block of yellow threads corresponds to processing unit 0; similarly for each additional block of threads.

[2] This isn't always the case in OpenMP programs– you can know just what you are and execute a section of code accordingly.


Blocks of Threads
Processing Unit Assignment
0
1
2
3
index
0
1
2
3
4
5
6
7
Array
A
B
C


1- dimensional Grid of Blocks of Threads,
where number of threads per block,
blockDim.x = 2
Within block,
threadIdx.x =
0 1
Thread Block num:
blockIdx.x =
0
1
2
3
Processing Unit Assignment
index
0 1 2 3 4 5 6 7
Array
A
B
C

available for the complex cases when our data would naturally map to them). The three variables that we can access in CUDA code for this example shown above are:

1. `threadIdx.x` represents a thread's index along the x dimension within the block.
2. `blockIdx.x` represents a thread's block's index along the x dimension within the grid.
3. `blockDim.x` represents the number of threads per block in the x direction.

In our simple example, there are a total of eight threads executing, each one numbered from 0 through 7. Each thread executing code on the GPU can determine which thread it is by using the above variables like this:

```
int tid = blockDim.x * blockIdx.x + threadIdx.x;
```

5.2 The Host and the Device

CUDA programs execute code on a GPU, which is a co-processor, because it is a device on a card that is separate from the motherboard holding the CPU, or central processing unit. In CUDA programming, the primary motherboard with the CPU is referred to as the *host* and the GPU co-processor is usually called the *device*. The GPU device has separate memory and different circuitry for executing instructions. Code to be executed on the GPU must be compiled for its instrution set.

5.3 CUDA Code for Vector Add

The overall structure of a CUDA program that uses the GPU for computation is as follows:

1. Define the the code that will run on the device in a separate function, called the *kernel* function.
2. **In the main program running on the host's CPU:**
   (a) allocate memory on the host for the data arrays.
   (b) initialze the data arrays in the host's memory.
   (c) allocate separate memory on the GPU device for the data arrays.
   (d) copy data arrays from the host memory to the GPU device memory.
3. On the GPU device, execute the *kernel* function that computes new data values given the original arrays. Specify how many blocks and threads per block to use for this computation.
4. After the *kernel* function completes, copy the computed values from the GPU device memory back to the host's memory.

Here is the actual code example for 8-element vector addition, with these above steps numbered. This is in the file **VectorAdd/CUDA/VA-GPU-simple.cu** in the compressed tar file of examples that accompanies this reading.

```
/*
 * A simplified example of vector addition in CUDA to illustrate the
 * data decomposition pattern using blocks of threads.
 *
 * To compile:
 *   nvcc -o va-GPU-simple VA-GPU-simple.cu
 */

#include <stdio.h>

// In this example we use a very small number of blocks
// and threads in those blocks for illustration
// on a very small array
#define N 8
#define numThread 2 // 2 threads in a block
#define numBlock 4  // 4 blocks

/*
 * 1.
 *  The 'kernel' function that will be executed on the GPU device hardware.
 */
__global__ void add( int *a, int *b, int *c ) {

    // the initial index that this thread will work on
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    // In this above example code, we assume a linear set of blocks of threads in the 'x' dimension,
    // which is declared in main below when we run this function.

    // The actual computation is being done by individual threads
    // in each of the blocks.
    // e.g. we use 4 blocks and 2 threads per block (8 threads will run in parallel)
    //      and our total array size N is 8
    //      the thread whose threadIdx.x is 0 within block 0 will compute c[0],
    //          because tid = (2 * 0)  + 0
    //      the thread whose threadIdx.x is 0 within block 1 will compute c[2],
    //          because tid = (2 * 1) + 0
    //      the thread whose threadIdx.x is 1 within block 1 will compute c[3],
    //          because tid = (2 * 1) + 1
    //
    //     The following while loop will execute once for this simple example:
    //          c[0] through c[7] will be computed concurrently
    //
    while (tid < N) {
        c[tid] = a[tid] + b[tid];       // The actual computation done by the thread
        tid += blockDim.x;       // Increment this thread's index by the number of threads per block
                                 // in this small case, each thread would then have a tid > N
    }
}


/*
 * The main program that directs the execution of vector add on the GPU
 */
int main( void ) {
    int *a, *b, *c;               // The arrays on the host CPU machine
    int *dev_a, *dev_b, *dev_c;   // The arrays for the GPU device

    // 2.a allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // 2.b. fill the arrays 'a' and 'b' on the CPU with dummy values
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i;
    }

    // 2.c. allocate the memory on the GPU
     cudaMalloc( (void**)&dev_a, N * sizeof(int) );
     cudaMalloc( (void**)&dev_b, N * sizeof(int) );
     cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    // 2.d. copy the arrays 'a' and 'b' to the GPU
     cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice );
     cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice );

    // 3. Execute the vector addition 'kernel function' on th GPU device,
    // declaring how many blocks and how many threads per block to use.
    add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );

    // 4. copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),
                              cudaMemcpyDeviceToHost );

    // verify that the GPU did the work we requested
    bool success = true;
    int total=0;
    printf("Checking %d values in the array.\n", N);
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
        }
        total += 1;
    }
    if (success)  printf( "We did it, %d values correct!\n", total );

    // free the memory we allocated on the CPU
    free( a );
    free( b );
    free( c );

    // free the memory we allocated on the GPU
     cudaFree( dev_a );
     cudaFree( dev_b );
     cudaFree( dev_c );

    return 0;
}
```

5.4 CUDA development and Special CUDA Syntax

The CUDA compiler is called **nvcc**. This will exist on your machine if you have installed the CUDA development toolkit.

The code that is to be compiled by nvcc for execution on the GPU is indicated by using the keyword `__global__` in front of the kernel function name in its definition, like this on line 21:

```
 */
__global__ void add( int *a, int *b, int *c ) {
```

The invocation of this kernel function on the GPU is done like this in the host code on line 82:

```
// declaring how many blocks and how many threads per block to use.
add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );
```

Note how you set the number of blocks and the number of threads per block to use with the `<<< >>>` syntax.

5.5 Larger Problems

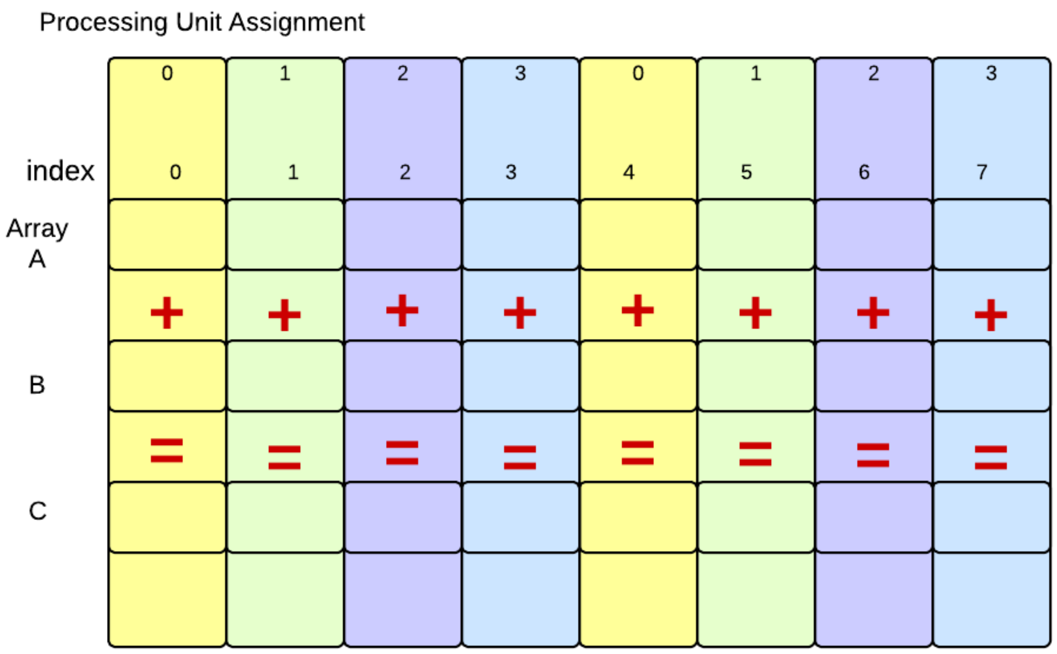
There is an additional file in the tar archive supplied for this reading that is called **VectorAdd/CUDA/VA-GPU-larger.cu**. You could experiment with larger arrarys in this version of the code. Comments within it explain how the thread assignment to array elements works if your array is larger than the number of total threads you use. CUDA and GPUs are better suited to much larger problems than we have shown for illustration here.

CHAPTER

SIX

OTHER WAYS TO SPLIT THE WORK

We have referred to the section of the array that a processing unit works on as a *chunk* of the array assigned to each processing unit. The example we have presented here divides the original and result arrays into equal-sized contiguous chunks, where the size of each chunk is the number of elements divided by the number of processing units (n/p). As you can imagine, there are other ways that the work could be divided. In fact, in the case of the CUDA example, each GPU thread worked on one element (the thread block was analogous to our processing unit). Chunks of size one are one possible alternative: every processing unit could work on elements like this:



This type of decomposition might work fairly well for shared-memory multicore computers using OpenMP, but it doesn't make as much sense for distributed systems using MPI, where the data must be sent from the master process to the other processes in the cluster– it is much easier to send a consecutive chuank at one time that small lieces over and over again.

Another alternative is to choose a chunk size smaller than n/p and each processing unit will work on then next available chunk. You can explore these alternatives in OpenMP by looking at documentation for the **schedule** clause of the pragmaa 'omp parallel for'. The CUDA code example called **VectorAdd/CUDA/VA-GPU-larger.cu** explores this concept.

6.1 Questions for reflection

In what situations would MPI on a cluster of computers be advantageous for problems requiring data decomposition?

In what situations would CUDA on a GPU be advantageous for problems requiring data decomposition?

In what situations would OpenMP on a multicore computer be advantageous for problems requiring data decomposition?

In multicore machines, the operating system ultimately schedules threads to run. Look up what default scheduling of threads to chunks is used in OpenMP if we leave out the **schedule** clause of the pragmaa 'omp parallel for'. Can you find any information or think of why this decomposition is the default?