

# Partitioned Memory Parallel Programming Framework

Tarun Beri, Sorav Bansal, and Subodh Kumar

Indian Institute of Technology, Delhi

**Abstract.** We present a framework for parallel programming. It consists of a distributed shared memory based simplified programming model, which leaves the application developer to focus mainly on task decomposition. This is a unified model for many-core processors (e.g., CPUs and GPUs), multiple processors on a system, as well as multiple systems. We also present a library implementation as a proof of concept of the model. It efficiently maps tasks to multiple compute engines, performs the required communication and schedules tasks to completion. In addition to convenience, the framework provides a race free programming environment by letting tasks own a partition of the memory. This simplifies programming significantly. We report a number of experiments.

## 1 Introduction

Hardware trends have necessitated a shift towards parallel software. Modern hardware environments have multiple compute cores on a single processor chip, multiple processors in one system, and multiple systems connected by a network. Moreover, each system may be equipped with not only general-purpose computing units (CPUs) but also special-purpose co-processors like graphics processing units (GPUs). The complexity and heterogeneity of this hardware environment presents a significant hurdle in efficient and intuitive programming.

Parallel programs are generally much harder to code and debug than serial programs. Variability of hardware architecture, performance tradeoffs and scalability issues also are much more complex. Issues like synchronization, scheduling, efficient data movement, and load balancing only complicate the job further.

We propose a new parallel programming model that hides much of this complexity and presents a simple way to program and reason. It is based on distributed shared memory model and proposes an explicit *shared-nothing* partitioning of read-write and read-only address spaces – each task executing independently in its own address space. Synchronization is limited to user defined sync-points, quite like the bulk-synchronous parallel model (BSP) [31]. Each task *owns* a section of the shared memory. Between sync-points it may write only to its portion. At each sync-point, the entire shared memory snapshot is effectively made available to each task in its local memory. The effect of “owner writes” and “sync-points” is that sequential consistency is trivially maintained and there are no race conditions or deadlocks among tasks to worry about. As a result, programming is simpler. And, as we demonstrate, programs remain efficient.

Our programming model does not distinguish between local tasks (running on the same host) and remote ones. Work sharing among tasks is controlled by the programmer in a portable manner so that the same program may run on many different configurations. The synchronization points are implicit in the program structure and clearly defined. Additionally, the program may control the synchronization method as well. Inter-task communication logically occurs only at these sync-points as in BSP.

As we show later through various examples and experiments, it is easy for a programmer to achieve parallelism at low synchronization costs by using well-designed custom synchronization methods. We also discuss some runtime optimizations that enable us to implement fast synchronization.

Our approach is the first comprehensive framework for achieving effective utilization of heterogeneous devices (CPUs and GPUs) connected locally or remotely. We demonstrate the generality and performance of our approach on a diverse set of applications running in a fairly complex hardware environment.

Our programming model is called BSP-RAMP (named after the Bulk Synchronous PRAM model [30] with the last ‘P’ for partitioned memory). We have implemented a library and a runtime environment based on this model in C/C++ called `pmlib`. `pmlib` provides primitives to hide the complexity and heterogeneity of the hardware environment and exposes the convenient and uniform abstraction of a task. The programmer provides subroutines that the tasks execute in isolated address spaces before synchronizing. The programmer is provided with an interface to specify how the memory is partitioned before the tasks start, and how they need to be merged back at task completion. The runtime environment schedules the tasks, moves the required data to these tasks before starting them, load-balances tasks on computing devices, and finally calls the synchronization routines at sync-points or after the tasks have finished.

The primary contribution of this paper is a simple yet expressive parallel programming model and a system supporting this model in heterogeneous environments. We have implemented scientific applications like  $N$ -body Simulation and Fast-Fourier Transform, commercial applications like Stock Pricing, data analytic applications like Page Ranking, and utility applications like Data Compression. With a porting effort of only around 2-3 man months, we could parallelize all these applications on an 8-node CPU/GPU cluster using `pmlib`. An attractive feature of `pmlib` is that it is trivial to parallelize many data-parallel applications. `pmlib` supports tasks that may be split among multiple threads. This reduces the simplicity of programming, but is included for generality. The API only supports synchronization among tasks. Intra-task synchronization is programmer’s responsibility.

The rest of the paper is organized as follows. Section 2 discusses related work, Section 3 discusses the programming model. Section 4 discusses our implementation, Section 5 discusses our experiments and results, and finally Section 6 concludes.

## 2 Related Work

Parallel programming models have been the subject of much research in past years. The holy grail is to let the programmer focus on program logic and have the compiler infer parallelism and generate parallel code. This remains challenging.

Two popular parallel programming frameworks are OpenMP[10] (shared memory) and MPI[17, 16] (message passing). The primary advantage of thread-based shared-memory programming is its intuitive programming model and fast interprocess communication. However, problems like data races, atomicity violations, and deadlocks cause much pain. On the other hand, message-passing programs eliminate these problems by having separate address spaces and exchanging messages for interprocess communication. Writing a program in this style often requires complex reasoning about the various states of the program.

A new programming architecture for parallel programming called CUDA[24] has emerged for highly data-parallel processing on GPUs (or CUDA devices). GPUs im-

plement hardware threads that can be created and scheduled quite fast (a few cycles compared to a few thousand cycles in CPUs). Hence, CUDA programs often use thousands of threads to perform a data-parallel task. The memory hierarchy (private and shared) on a GPU is fully exposed to the programmer for program-specific data placement and optimization. A CUDA program requires a *host* CPU. The device memory is independent of the host memory and user must explicitly copy the data.

The three programming models, OpenMP, MPI, and CUDA, are quite contrasting in the way they deal with parallelism. Clearly each model is best suited for a certain environment: OpenMP for shared-memory multiprocessors, MPI for distributed computing, and CUDA for highly data-parallel compute devices. We unify these models. This unified model frees to the programmer to design the parallelism, while underneath we use a combination of all these models to provide an efficient implementation. Our unified abstraction is general enough to express many different types of programs without losing performance. Our abstraction is also intuitive and minimal, allowing the programmer to focus on the logic.

Many other approaches to parallel programming exist. Some use programming language constructs that allow the programmer to express parallelism. Split-C[23] proposes interesting extensions to the C programming language such as the ability to declare an array spread across multiple processors, loop parallelization, bulk assignments, and the `atomic` keyword. Cilk[18] allows parallelization of a program at the function call boundary by adding language constructs to allow each function call to spawn a separate thread. Similar, yet subtly different, approaches exist for other languages such as Fortran (High-Performance Fortran[21], Co-Array Fortran[26]), Java (Titanium[19]). Some other programming extensions to C/C++ have been proposed in Unified Parallel C[13] and Z-level programming language[9], both aimed at allowing expression of data-level parallelism on arrays. Chapel[22] is a new imperative-style parallel programming language, with many of its parallel features influenced by Z-level programming language and High-Performance Fortran. OpenMP is also a language-based approach. Most language-based approaches use shared address spaces and fine-grained synchronization. Hence, usual problems like race conditions and deadlocks still remain. Some of these approaches use a custom runtime environment to schedule and load-balance the multiple threads, while others simply use threads underneath and then rely on the OS to schedule them.

A complementary approach to specifying parallelism is to provide a library with ‘utility’ functions that helps the programmer express parallelism. Parallel Virtual Machine[8] has a notion of decomposition into tasks (usually implemented using processes) that use message-passing. PVM supports heterogeneous operating systems, networks, and applications. GLOBUS[14] is a similar initiative to program a “grid” of computers. BSPlib [20] implements the BSP model in terms of local processes with a powerful but complex registration protocol. Similarly, MPI is also library-based. Although language based approaches are usually more succinct and, maybe, more satisfying, library based approaches are simpler to implement and more easily extended. Hence, they are more popular.

Data communication and synchronization is a critical component of any parallel programming framework and boasts a large body of research including those that provide a shared view of the distributed memory, e.g, global arrays[25], SHMEM[28] and PAWS[7]. PAWS provides a flexible framework to allow distributed applications to share parallel data structures. Applications directly need to specify synchronization

and are able to dynamically create “connections”. PADRE[11] is another framework for parallel data routing.

Others like POOMA[29] and Overture[5] focus on scientific computations and provide inbuilt grid partitioning strategies. TULIP[6] is a powerful object-parallel distributed programming tool for remote method execution and remote load/store operations. Our approach borrows many ideas from existing ones. `pmlib` is built on top of OpenMP, MPI, and CUDA. Its novelty is in the interface, which is general and intuitive, yet efficient. It supports a wide variety of applications but it also supports a wide variety of environments. To our knowledge, this is the first attempt to unify CPU and GPU computation into a single programming model. As we elaborate later, our unified interface is not at the cost of efficiency. The `pmlib` library is accompanied with a runtime environment that handles runtime issues like scheduling, load balancing, data distribution and synchronization.

### 3 Programming Model

A sequential program may employ our library to execute all or part of its functionality in parallel. To do so, our library provides the following constructs:

**Task:** A task is a construct to parallelize a portion of the computation. For example, a task that needs to be parallelized *globally* across a cluster of hosts can be specified with the construct `globalTask`.

**Subtask:** A subtask is a portion of the task that can be executed in parallel with other subtasks. A subtask computation is specified with a *subtask callback* function. This function is specified in the task construct.

**RO-segment:** This is a read-only data segment that is logically visible to all subtasks.

**RW-segment:** This is a read-write data segment that is logically visible to all subtasks. Each subtask has an exclusive copy of RW-segment to work with and the results of all subtasks are synchronized (merged) at subtask or task boundaries (sync-points).

**Pre-subtask and Post-subtask Callbacks:** These are optional callbacks which are made respectively before and after the completion of subtask function. Pre-subtask callback is typically used by the programmer to specify the distribution of input data (RO and RW) and post-subtask callback defines the synchronization methods of the output (RW) produced by all subtasks.

**Reduce Callback:** This optional callback supports reduction of data in memory segments using associative reduce operations. In particular, it supports local computation across tasks on a single system before sending the intermediate results on the network, to be globally reduced. This construct is especially useful for Map-Reduce style computations.

Because tasks actually execute in separate address spaces (exclusive RW copies) and explicit functions are used for data synchronization (pre-subtask/post-subtask), our programming model is free of race conditions and deadlocks. Further, pre-subtask and post-subtask callbacks allow extremely flexible data distribution and synchronization. This is in contrast with another BSP-like approach used in Determinator[4] where data is merged at pre-specified (doubleword) granularity.

The application programmer is relieved of the syntax to deal with thread creation, management and destruction. Also, many GPU complexities (like CUDA kernel invocation, data transfers between the host and the device) are hidden from the programmer. A serial C/C++ program can be *ported* to `pmlib` framework trivially.

```

void serialMatrixMultiply(int* mA, int* mB, int* mC,
                        int dim1, int dim2, int dim3) {
    for (i = 0; i < dim1; i++)
        for (j = 0; j < dim3; j++)
            for (k = 0; k < dim2; k++)
                mC[i*dim3 + j] += mA[i*dim2 + k] * mB[k*dim3 + j];
}

```

**Fig. 1.** Serial Matrix Multiplication Routine. mA, mB are the input square matrices of dimensions  $\text{dim1} \times \text{dim2}$  and  $\text{dim2} \times \text{dim3}$  respectively. mC is the output matrix with dimensions  $\text{dim1} \times \text{dim3}$

```

/* Programmer Invariants: No. of rows is divisible by no. of subtasks
   Matrices A, B are placed consecutively in RO memory region. */
status matrixPostSubtask(memoryPartition pMem, taskConfig pConf) {
    int sM = pMem.lengthRO/(2 * sizeof(int));
    int dim = sqrt(sM); /* square matrix */
    int rowSlice = dim / pConf.subTaskCount;
    /* Data Synchronization is done in scatter gather style
       Parameters: Data Segment, Sync Length, Offset, Count, Jump */

    syncDataSegment(pMem.dataRW, rowSlice*dim, 0, 1, 0);
    return success;
}

status matrixPreSubtask(memoryPartition pMem, taskConfig pConf) {
    int sM = pMem.lengthRO/(2 * sizeof(int));
    int dim = sqrt(sM); /* square matrix */
    int rowStart = pConf.subTaskId * rowSlice;

    resetDataSegment(pMem.dataRW, pMem.dataRW+rowStart*dim);
    return success;
}

status taskMatrixMultiply(memoryPartition pMem, taskConfig pConf) {
    int sM = pMem.lengthRO/(2 * sizeof(int));
    int dim = sqrt(sM); /* square matrix */
    int rowSlice = dim / pConf.subTaskCount;
    int rowStart = pConf.subTaskId * rowSlice;

    serialMatrixMultiply(pMem.dataRO+rowStart*dim, pMem.dataRO+sM, pMem.dataRW,
                        rowSlice, dim, dim);
    return success;
}

void parallelMatrixMultiply(void) {
    setPreSubtaskCallback(matrixPreSubtask);
    setPostSubtaskCallback(matrixPostSubtask);
    globalTask(inputMem, inputLength, outputMem, outputLength, taskMatrixMultiply);
}

```

**Fig. 2.** Parallel Matrix Multiplication Routine using pmlib. The function serialMatrixMultiply is the same as the serial version except it only multiplies the elements of mA and mB for mA's rows rowStart..rowStart+rowSlice

Parallelizing the program however requires more work. However, our approach allows the programmer to parallelize sections of the serial program incrementally.

To understand the programming model better, let us consider a simple example on matrix multiplication. A simple serial matrix multiplication routine is shown in Figure 1, and a corresponding parallel matrix multiplication routine using pmlib is shown in Figure 2. The matrix multiplication routine is specified as a task divided into a number of parallel subtasks. The subtask callback is not very different from the serial code except that it operates only on a subset of the rows of the output matrix. The specification of the individual data regions of the subtasks is done in the pre-subtask callback, which uses the subtask-id to distinguish among multiple subtasks. The post-subtask callback synchronizes the results computed (RW segment) by the subtask to the master's copy. The programmer must implement both CPU and GPU subtask callbacks. For brevity, we only use the CPU subtask callback in this example. We next describe some more subtle features of pmlib.

**Callback Chains** Most applications can be efficiently parallelized using the interface described above. Some applications require more. For example, a programmer may wish to execute one callback after another (with sync-points in between) on each subtask. Our library allows the programmer to specify a sequence of callbacks, called callback chains. A chained task is an efficient replacement for multiple tasks which

intend to work with the same input data. In particular, it avoids multiple transfers of RO-segment from the master (the host where program is started) to other nodes. Chained task also reduces the overhead of subtask creation and destruction multiple times.

For chained tasks, our library allows another data array called the WO (write-only) data section. The WO section is the private memory of a subtask and is passed from one callback to the next callback in the callback chain. There exists a sync-point at the end of every callback in the chain. This automatically enforces a barrier between every two subsequent callbacks and thus all subtasks at any point in time are guaranteed to be executing the same callback in the callback chain.

**Cycling Subtasks** We allow the programmer to run a subtask multiple times before returning back to the master thread. This feature is useful for applications which perform multiple iterations of the same core logic. Another important utility of this feature is to program devices with limited memory (e.g., CUDA devices) where the same subtask callback is required to be called for different data. Different iterations of the callback can load different data from main memory in such situations. When subtask cycling is enabled, the pre-subtask and post-subtask callbacks are also executed on each iteration.

**Callback Rebounds** Sometimes, the program requires all subtasks to execute repeatedly in an iterative manner with a sync-point after each iteration. This is similar to chained tasks, except that we can now support a variable number of iterations of the same callback function. Algorithms that require iterative convergence greatly benefit from this feature.

## 4 Implementation

We have implemented our library `pmlib` on Windows XP and Ubuntu Linux 8.04.2. On Windows XP, we use CL compiler (VS 2005) which supports OpenMP version 2.0 and on Linux, we use gcc 4.2 which supports OpenMP version 3.0. For GPU computations, we use CUDA version 3.1. On Windows XP, we use Microsoft Compute Cluster Pack[2] and on Linux we have tested our implementation with MPI-CH and OpenMPI.

The library is initialized by spawning MPI processes on all hosts. As with MPI norms, the first host becomes master and all others become slaves. Each of these processes call the library's `initialize()` function. `initialize()` effects a handshake and control setup routine, where the master and all slaves exchange version and other control information. The slaves inform the master about the number and type of their local CPU and GPU processing elements. `initialize()` also sets up a local task scheduler on each of the hosts. After initialization, the task schedulers on all slave hosts communicate with the one on master to distribute and execute parallel tasks.

Once `pmlib` is initialized on all hosts, the master host can create a parallel task (using `globalTask` or other library constructs). The programmer specifies the number of CPU and GPU subtasks he would like to spawn or the library automatically creates one subtask per processing element. The programmer also specifies the data arrays, subtask callbacks, and the optional pre and post subtask and reduce callbacks.

Once a parallel task is created, the library operates in three stages: data distribution, parallel subtask execution, and finally data collection and synchronization. Division and movement of data are crucial to the performance of any parallel programming model. In `pmlib`, the division and placement of data in RO and RW sections is application dependent (through pre and post-subtask callbacks).

The reduce callback is implemented on RW-data section of all subtasks in a tree-like manner to form the master’s copy of the reduced data. Because the library provides no guarantees on the order of the reduction, we assume that the reduction operator is associative. The library also provides some pre-implemented reduce callbacks like `reduceAdd`, which stands for reduction through addition.

Our library handles data distribution and synchronization across the network by making copies of the required extents of RO and RW segments on each host. For efficiency, the library internally performs two-stage data synchronization. All subtasks within a host synchronize among themselves to the local host’s memory copy in the first stage. This can happen in parallel. In the second stage, all slave hosts in the cluster synchronize to the master host over the network.

All CPU subtasks running on a host share the same RO segment but a separate copy of RW segment is created for each subtask. Copies of both RO and RW segments are made on GPU memory for GPU subtasks.

For applications requiring large data transfers on the network, our library supports compression. It has in-built standard zlib-like[1] compression and also allows the programmer to implement custom compression. The same interface also supports data encryption.

## 5 Experimental Results

Our experiments were performed on a cluster of four 64-bit eight core machines with Intel Xeon E5450 3.00 GHz CPU running Ubuntu Linux 8.04.2 connected using a 1Gbps ethernet network. Two of the hosts also included a Tesla C1060 card each.

For our experimental results, we discuss examples from various application domains. We start with a simple array summation example. Then, we discuss three scientific applications - Matrix Multiplication,  $N$ -body Simulation and Fast Fourier Transformation. Next, we discuss a commercial application that estimates stock pricing based on Monte Carlo equations. This is followed by an implementation of PageRank[27], which is a popular example based on MapReduce-style programming[12]. Finally, we discuss a zlib based compression program. Of these examples, PageRank and Fast Fourier Transform are highly data intensive i.e., they have a low compute-to-communication ratio.

Our experiments evaluate the efficiency, scalability, generality and intuitiveness of our programming model. We find that our model is capable of effectively parallelizing the workload across many hosts and devices with little programmer effort. For many of our workloads, we observe linear (sometimes super-linear due to memory effects) scaling of performance with the number of processing elements.

For most experiments, writing GPU CUDA kernel was easy with the exception of PageRank whose CUDA kernel uses global memory atomic operations and is thus less efficient. Writing optimized CUDA kernel is often tricky and time consuming and it also requires a lot of careful thought over many parameters like organization of CUDA threads into blocks and blocks into grids, efficient use of memory hierarchy (especially shared memory) and the vast number of registers on CUDA’s SMs (Streaming Multiprocessors). We wrote efficient kernels as much as possible. In our experiments, only a single subtask is assigned to the GPU.

In this paper, we report results when parallelizing exclusively on CPUs or exclusively on GPUs. We have also extensively tested our library with parallelization across both CPUs and GPUs simultaneously. The results of simultaneous execution on CPUs and GPUs are usually similar to those obtained when executing on the

slower processing element, given the wide performance difference between the two. Refer to our technical report[3] for detailed results.

### 5.1 Summation

This experiment tests the performance of `pmlib` on a linearly parallelizable problem. It repeatedly computes (25 times for our experiments) the following in-place function on each element of an integer array and then finds the sum of all elements of the resulting array.

$$f(x) = \left[ C \times (\sin x)^{\cos \frac{1}{3} x} \right] \tag{1}$$

Here,  $x$  is an element of the integer array and  $C$  is an integer constant. To parallelize, each subtask applies the function  $f(x)$  in parallel to different parts of the input array and then computes its sum. Finally, all subtasks add their results to find the total sum using the reduce callback. The parallel implementation using our library achieves a speedup of around 54x over a serial implementation for an array of 60 million elements. This speedup is obtained if the computation is parallelized across GPU cores. When parallelizing only on CPUs, we obtain 17x speedup over serial implementation. The high speedup when using GPUs is primarily due to their relatively high compute power: the GPU subtask uses thousands of simultaneous threads.

### 5.2 Square Matrix Multiplication

Our second application multiplies two square matrices of large dimensions. To parallelize the routine, we divide the rows of the first input matrix equally among subtasks. Each subtask sees the full second input matrix, and computes its part of the result. We wrote a CUDA kernel to assist with matrix multiplication. Our kernel is simple: it uses only the device’s (slow) global memory and ignores its (faster) shared memory. A more efficient CUDA kernel that exploits GPU’s memory hierarchy better is likely to scale even better.

We present our results in Table 1. For the largest matrix size of  $5000 \times 5000$ , we obtain a speedup of about 75x. In general, the speedup increases with increasing matrix size, as more computation is done per unit of communication for larger matrices due to the  $O(n^3)$  nature of the algorithm.

Square Matrix Dimension	Time (seconds)					
	Serial	Parallel over CPUs (1 host)	Parallel over CPUs (cluster)	Parallel over GPUs (1 host)	Parallel over GPUs (cluster)	
100	0.001	0.001	0.02	0.53	0.54	
500	0.2	0.1	0.1	0.5	0.6	
1000	5.2	0.8	0.6	0.7	0.7	
2000	45.9	6.2	3.5	1.6	1.6	
3000	172.2	25.6	11.2	4.3	3.5	
4000	433.0	65.6	22.3	9.5	6.9	
5000	919.2	126.2	42.7	18.0	12.3	

**Table 1.** Results of matrix multiplication experiment

For matrix dimensions less than  $500 \times 500$ , the performance on parallelizing on a single host is faster than the performance of parallelizing across a cluster of hosts (using only CPUs). In case only GPUs are used, parallelizing on a single host is faster than parallelizing on multiple hosts, up to matrix dimension  $1000 \times 1000$ . Using multiple hosts over ethernet helps only when the problem size is large enough so that the benefits of parallel computation outweigh the communication costs.

We also study the performance of `pmlib` with increasing number of hosts. Table 2 shows the results of matrix multiplication experiment using CPU cores of 1, 2, 4 and 6 hosts respectively. Each host used in this experiment has an 8-core CPU. As expected, the parallel performance on `pmlib` scales with the number of hosts at high matrix dimensions.

Square Matrix Dimension	Matrix Multiplication Time (seconds)			
	Parallel over CPUs (8 CPU cores per host)			
	1 host	2 hosts	4 hosts	6 hosts
2000	6.1	4.2	3.5	4.0
3000	25.6	15.2	11.2	10.5
4000	65.6	39.7	22.3	21.7
5000	126.2	79.8	42.7	39.2
10000	1757.5	1131.8	571.8	319.2

**Table 2.** `pmLib` performance with increase in number of hosts

### 5.3 N-body Simulation

This experiment is a simulation of a system of particles (or physical bodies having mass), which exert mutual gravitational pull on each other. The motion of each particle is governed by the resultant force from all other bodies in the system. We use a brute force  $O(n^2)$  approach to compute gravitational forces between every pair of particles. Using `pmLib` we equally partition the work among all available processing elements. We used the kernel from CUDA SDK version 2.3 for this experiment. On GPU cluster, the parallel version of the benchmark is 326x faster than the serial version for a simulation of 50,000  $N$ -bodies with 100 iterations. As with the matrix multiplication example, the advantage of using parallel processors increases with increasing computation. For a system of 100,000  $N$ -bodies, the speedup (for 100 iterations) over serial implementation is 100253x! This astonishing result is due to the large number of GPU threads running in parallel.

### 5.4 Fast Fourier Transform

Our next experiment parallelizes a well-known discrete fourier transform algorithm, called the Fast Fourier Transform (FFT). We parallelize the computation of 2-D FFT for a matrix of randomly generated time domain complex numbers. To parallelize, we first compute 1-D FFT over all rows of the matrix (in parallel) and then compute 1-D FFT over all columns of the resulting matrix (also in parallel). The 1-D FFT computations on rows and columns is separated by a sync-point (one task finishes and another starts). We use a CUDA kernel available in the publicly available CUFFT[15] library for this experiment. Due to the large amount of data transfer involved in this experiment, the communication cost far exceeds the computation time. As a result, computation is fastest when done on one host rather than on the cluster. For an input matrix of dimension  $2^{12} \times 2^{15}$ , the time taken for 2-D FFT on one host is 8.79 seconds, while on 4 hosts it is 35.67 seconds. The amount of data transferred in this process is more than 3.0 GB, which on our 1Gbps network takes about 31.5 seconds. So the overall computation time spent by each host in parallel setup is less than 3s (this time includes the library’s overheads – memory allocation, buffering, thread management and scheduling). We expect this experiment to scale better on multiple hosts if a faster network (e.g., 100Gbps ethernet) is available. On our experimental setup, we achieve a speedup of 4.63x by parallelizing on 8 CPU cores of a single host.

### 5.5 Monte Carlo Stock Pricing

Monte Carlo Stock Pricing is a method of computing the expected future price of a set of stock options by considering variations along many possible paths, each of which is randomly chosen using a normal distribution. The price of the stock is assumed to follow a stochastic differential equation on each path. The final price is the average of expected prices over all paths. This process is repeated for a large number of independent stock options. We parallelize different stock options using different subtasks. We used a kernel from CUDA SDK version 2.3 for our GPU implementation of the application. For our GPU cluster, we achieve a speedup of 2517x over serial implementation for 1024 options considering variations along  $2^{26}$  paths on our experimental setup. Once again, the high speed is due to GPU’s high suitability to such applications.

## 5.6 Page Rank

PageRank is an example of a data-analytic algorithm that computes the search rank of a webpage based on the web’s hyperlink structure. The search rank of a webpage is the probability of a random surfer visiting it. The algorithm works by first uniformly initializing the ranks of each page to a constant value, and then iteratively transferring the ranks of all web pages to their outlinks till the ranks of all pages converge (or till a maximum number of iterations). PageRank is a popular application of the Map-Reduce paradigm.

We generated random web graphs of different sizes and then tested our PageRank implementation on them. We also wrote a CUDA kernel for the PageRank algorithm, and ported our serial implementation to `pmlib`. Our CUDA implementation for the application is relatively dumb. The application has a large memory footprint and data sharing between CUDA threads. We again only use the GPU’s global memory and avoid race conditions among threads of the GPU subtask using slow global memory atomic instructions.

We implemented the iterations of the PageRank algorithm using callback rebounds discussed in Section 3. We used reduce callback to add the contributions to a page’s rank from all its inlinks (just like this is done in Map-Reduce). We perform 100 iterations of this algorithm to arrive at a PageRank.

Web Size (in million pages)	Page Rank Computation Time (in s)				
	Serial Task	Parallel over CPUs (1 host)	Parallel over CPUs (cluster)	Parallel over GPUs (1 host)	Parallel over GPUs (cluster)
1	15.7	10.8	9.6	46.2	43.2
5	197.2	125.0	56.0	95.1	77.1
10	428.3	234.2	112.0	159.4	123.4
15	655.0	346.1	176.6	259.6	187.3
20	850.3	464.8	237.0	375.1	260.7
25	1076.5	524.3	298.2	484.2	322.9
30	1293.0	636.5	380.7	622.1	408.1

**Table 3.** Results of page rank experiment

PageRank being a data-intensive application causes communication cost to dominate. Our implementation’s data compression feature (during transfer) significantly reduces this communication cost.

We tabulate our results in Table 3. Like FFT, computation of PageRank is faster on CPU cores than on CUDA devices. However, the PageRank’s computation-to-communication ratio is better than that of FFT. Hence, we obtain performance improvements by parallelizing over multiple hosts. Our parallel implementation is 3.4x faster over serial implementation for a web size of 30 million pages. The bottleneck in this experiment is always the data transfer time, and a faster network will improve results.

## 5.7 Data Compression

Finally, we evaluate our library on in-memory data compression using `zlib`. A file is read into memory and then `pmlib` is used to compress different blocks (of the file) on different processing elements in parallel. We have not written (or otherwise obtained) a CUDA kernel for this experiment yet. Hence, we only report results on parallelizing over CPUs. Using the 8 CPU cores of 1 host, our parallel program achieves a speedup of 7.63x (over serial implementation) while compressing a 1GB text file.

## 6 Conclusions and Future Work

`pmlib` provides a unified framework for programming CPUs and GPUs on a cluster of machines. It allows the programmer to decompose computation into tasks, and then

provides a runtime environment to efficiently schedule, distribute data, and synchronize these tasks.

Although much work remains on the implementation front, early experience from the model is highly encouraging. We implemented a variety of applications in this model using `pmlib` and found that the programming overhead of the parallel implementation over the sequential one was practically negligible. We strongly believe the simplified model, where the task owns a section of the (distributed) shared memory, is the main reason for this. Indeed, most debugging can be done in the serial version itself. We feel a number of parallel applications, especially data-parallel ones, naturally fit this paradigm. The model can simulate general BSP as well, so it is not a weak programming model.

Although we have not spent much time on the implementation, the speed-ups are appreciable. We do think, there is more improvement possible. We have not yet focussed on load balancing, which we propose is easy in this model. The library implementation can also be improved with some program analysis: there is scope to reduce the amount of memory copy performed at synchronization. We are also investigating process migration and fault tolerance.

## References

- [1] *A massively spiffy yet delicately unobtrusive compression library*, <http://www.zlib.net/>.
- [2] *Microsoft compute cluster pack*, [http://msdn.microsoft.com/en-us/library/cc136762\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/cc136762(VS.85).aspx).
- [3] *Pmlib - partitioned memory parallel programming library*, <http://www.cse.iitd.ac.in/~tarun/pmlib.html>.
- [4] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford, *Efficient System-Enforced Deterministic Parallelism*, OSDI '10: 9th USENIX Symposium on Operating Systems Design and Implementation, 2010.
- [5] Federico Basseti, David Brown, Kei Davis, William Henshaw, and Dan Quinlan, *Overture: An object-oriented framework for high performance scientific computing*, In Proceedings of SC98: High Performance Networking and Computing. IEEE Computer Society, 1998, p. 9.
- [6] Peter Beckman and Dennis Gannon, *Tulip: A portable run-time system for object-parallel systems*, Parallel Processing Symposium, International **0** (1996), 532.
- [7] Peter H. Beckman, Patricia K. Fasel, William F. Humphrey, and Susan M. Mniszewski, *Efficient coupling of parallel applications using paws*, HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (Washington, DC, USA), IEEE Computer Society, 1998, p. 215.
- [8] Arndt Bode, Jack Dongarra, Thomas Ludwig 0002, and Vaidy S. Sunderam (eds.), *Parallel virtual machine - europvm'96, third european pvm conference, münchen, germany, october 7-9, 1996, proceedings*, Lecture Notes in Computer Science, vol. 1156, Springer, 1996.
- [9] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder, *The high-level parallel language zpl improves productivity and performance*, In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing, 2004.
- [10] Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, October 2007.
- [11] Kei Davis and Daniel J. Quinlan, *The parallel asynchronous data routing environment padre*, Workshop on Object-Oriented Technology (London, UK), ECOOP '98, Springer-Verlag, 1998, pp. 454–455.

- [12] J. Dean and S Ghemawat, *Mapreduce: simplified data processing on large clusters*, Commun. ACM **51**, 1, Jan 2008, pp. 107–113.
- [13] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper, *Upc language specification v1.1.1*, (2003).
- [14] I. Foster, *Globus toolkit version 4: Software for service-oriented systems*, IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS, 2006, pp. 2–13.
- [15] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manfredelli, *High performance discrete Fourier transforms on graphics processors*, SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (Piscataway, NJ, USA), IEEE Press, 2008, pp. 1–12.
- [16] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres, *Open MPI: A flexible high performance MPI*, Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics (Poznan, Poland), September 2005.
- [17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing **22** (1996), no. 6, 789–828.
- [18] Supercomputing Technologies Group, *Cilk-5.3 reference manual*, Technical Report (2000).
- [19] Paul Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick, *Titanium language reference manual*, Technical Report CSD-01-1163 (2001).
- [20] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling, *BSPlib: the bsp programming library*, Parallel Computing **24** (1998), 1947–1980.
- [21] Rice University Houston, *High performance fortran language specification version 2.0*, Technical report (1996).
- [22] Cray Inc., *The Chapel language specification version 0.4*, Technical report (2005).
- [23] Arvind Krishnamurthy, David E. Culler, Andrea Dussseau, Seth C. Goldstein, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick, *Parallel programming in split-c*, In Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, 1993, pp. 262–273.
- [24] J. Nickolls, I. Buck, M. Garland, and K Skadron, *Scalable parallel programming with cuda*, ACM SIGGRAPH 2008 Classes **98** (2008), no. E2, 3247–3259.
- [25] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra, *Advances, applications and performance of the global arrays shared memory programming toolkit*, vol. 20, 2006, pp. 203–231.
- [26] Robert W. Numrich and John Reid, *Co-array fortran for parallel programming*, (1998), 17:1–31.
- [27] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, *The pagerank citation ranking: Bringing order to the web*, Technical Report (1999).
- [28] Parzyszek and Krzysztof, *Generalized portable shmem library for high performance computing*, Ph.D. thesis, Ames, IA, USA, 2003, Co-Major Professor-Kendall, Ricky A. and Co-Major Professor-Lutz, Robyn R.
- [29] John V.W. Reynders III and Julian C. Cummings, *The pooma framework*, Comput. Phys. **12** (1998), 453–459.
- [30] Alexandre Tiskin, *The bulk-synchronous parallel random access machine*, Euro-Par'96 Parallel Processing (Luc Boug, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, eds.), Lecture Notes in Computer Science, vol. 1124, Springer Berlin / Heidelberg, 1996, pp. 327–338.
- [31] Leslie G. Valiant, *A bridging model for parallel computation*, Commun. ACM **33** (1990), 103–111.