Institut für Parallele und Verteilte Systeme

Abteilung Parallele Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Master Thesis Nr. 3355

# Data Structures and Algorithms in Unified Parallel C for Molecular Dynamics

Kamran Idrees

| | |
|---|---|
| **Studiengang:** | M.Sc. Information Technology |
| **Prüfer:** | Prof. Dr.-Ing. Sven Simon |
| **Betreuer:** | Dipl.-Inf. Steffen Kieß |
| **begonnen am:** | 09.07.2012 |
| **beendet am:** | 17.10.2012 |
| **CR-Klassifikation:** | C.1.2, D.3.2, E.1 |

# Abstract

Partitioned Global Address Space (PGAS) is an abstraction of the shared memory model for Single Program Multiple Data (SPMD). PGAS was developed to unite the concepts of shared memory and distributed data into a single parallel programming model. The purpose of allying distributed data with shared memory was to cultivate locality-aware shared memory paradigm. In PGAS, shared space is partitioned among threads, such that each thread has a portion of shared space which is local to it. Each thread can exploit locality by effectively doing computation on data which has affinity to it or which resides in its local space.

Unified Parallel C (UPC) is a parallel extension of ISO C. It is distributed shared memory programming model which is based on PGAS to support SPMD programs. UPC aims to support locality-aware paradigm. The declarations in UPC provide control to the programmer to efficiently distribute data across multiple threads, which can later be manipulated by the threads such that each thread manipulates data which has affinity to it to exploit locality. However UPC does not restrict the access to data which has affinity to any other thread. Apart from shared space, each thread has also a private space which can only be accessed by thread which owns this space. UPC provides library functions for moving data to/from shared memory economically, efficiently dividing the tasks among the threads.

Molecular dynamics (MD) simulates interactions between molecules. In principle, given an initial set of positions and velocities of molecules, the following time progression of a set of interacting molecules is ascertain. After the system is initialized with the initial positions and velocities of molecules, calculation of forces is done on all molecules in system. Finally Newton's equations of motion are integrated to advance the positions and velocities of molecules. The simulation is advanced unless the computation of time evolution of system is completed for the aimed length of time.

We have ported an in-house MD code (CMD) to UPC, calculating interactions between the molecules and atoms in a parallel fashion. The molecules are spatially decomposed into cells which then are distributed among threads. Each thread calculates the interactions of the molecules of the cells it has affinity to. To minimize the access to remote data, the cells are distributed among the threads in a spatially coherent manner. We have tried to aid cache optimizations in most of the routines in our ported MD code. Here we elaborate the technical details of UPC which are necessary to understand our ported CMD code, then we explain the methodology we chose for porting our CMD code to UPC and finally we present the benchmark results for our UPC implementation of CMD.

# Table of Contents

# List of Figures & Examples

# 1 Introduction

## 1.1 Partitioned Global Address Space (PGAS)

It seems like shared memory parallel programs are easy to write compared to distributed memory programs [1], as access to any memory location is identical in shared memory paradigm because all threads view a single address space. Therefore there is not any restriction or resistance for a thread to access a data which is designated for some other thread to compute on. There is no locality-awareness in shared memory models, which might lead to unwanted proliferation of accesses to the data which is not local to a thread. This leads to different cached value of single memory location and therefore shared memory models rely heavily on a cache coherence system for maintaining the local copy of the data consistent. This results in deterioration of performance and scalability of an application written for a shared memory model [5]. Hence, shared-memory programming models fail to exploit locality effectively. This leads to the development of Partitioned Global Address Space.

Partitioned Global Address Space is a locality-aware distributed shared memory model for Single Program Multiple Data (SPMD) stream (discussed later in chap. 2). In this memory model, the shared address space is distributed among threads, thus named Distributed Shared Memory Model. Each thread has a portion of the address space local to it. Hence making it possible for mapping of each thread and data (which is local to this thread) to the same physical node. Programmers can therefore allocate the data to be computed on by a thread to the local space of the thread, which makes it locality-aware programming model. Each thread then knows the data which is local to it and thus majorly computes on this data, resulting in very less or no access at all to remote data. PGAS allows multiple threads to be working on data in the same or entirely different way, unlike data parallel model where multiple threads processes data in strictly identical manner [5]. Therefore PGAS model does not only provide ease of shared data programming but also abstain from uncompromising flow of data parallel model where multiple threads need to process data in similar way.

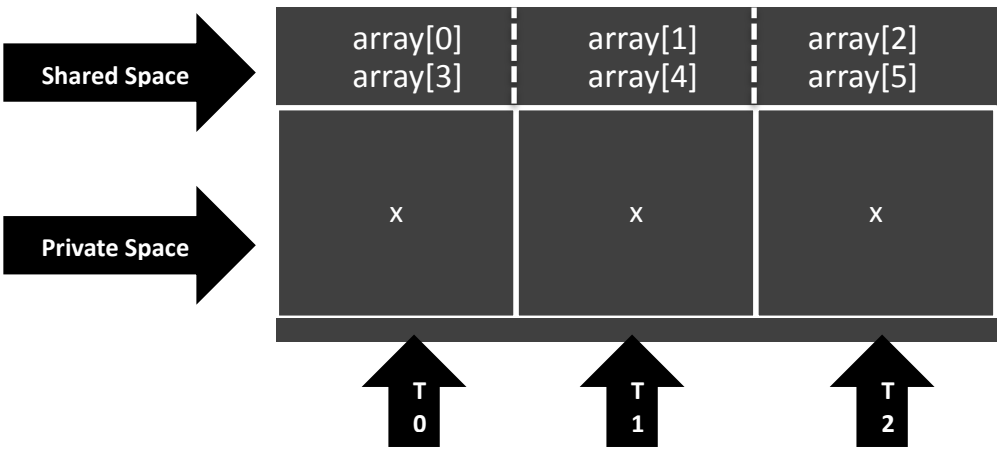


Figure 1.1: Partitioned Global Address Space

Consider an example of PGAS model in Figure 1.1. Each thread has local variable x in its private address space; therefore it can only be accessed by the thread who owns this variable. There is also an array which is allocated in the shared address space, having its elements distributed in round robin fashion across the shared space. Thus each thread has few elements of array in the fragment of shared address space which has affinity to it. Each thread can access all elements of array but to access elements of array which are in the portions of shared address space of other threads, a thread must use pointer to shared region. Whereas, each thread can exploit the locality by doing extensive computation on the elements of array which resides in its portion of shared address space. In figure 1, Thread 0 has a local variable x which can only be accessed by Thread 0. Thread 0 also have access to all elements of

shared array but it can exploit locality by computing extensively on the array elements which resides in its portion of shared address space (which are array[0] and array[3]).

## 1.2 Unified Parallel C (UPC)

UPC is an explicit parallel extension of C language, more specifically it is an extension of ANSI C99 standard, and therefore it inherits all the functionality available in ANSI C99 standard [1]. It is set up on Partitioned Global Address Space model, hence threads in UPC share part of their address space. Each thread has a portion of shared memory which is local to it. UPC provides data declarations and rich set of library functions to distribute the data among threads, divide the loop iterations or different tasks among threads and coordination among threads. A detailed description of UPC functionality is covered in chapters 2 and 4.

## 1.3 Molecular Dynamics

Molecular dynamics (MD) simulations calculates interactions between molecules. In principle, given an initial set of positions and velocities of molecules, the following time progression of a set of interacting molecules is ascertain. After the system is initialized with the initial positions and velocities of molecules, calculation of forces is done on all molecules in the system. Finally Newton's equations of motion are integrated to advance the positions and velocities of molecules. The simulation is advanced unless the computation of time evolution of system is completed for the aimed length of time. Figure 1.3 shows an image of a molecular dynamics simulation.

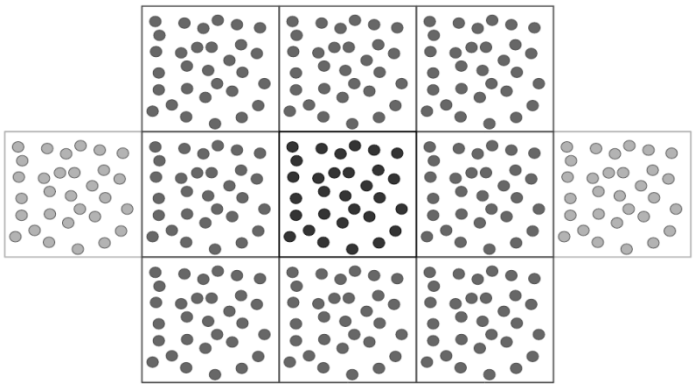

Figure 1.3: Molecular Dynamics Simulation

# 2 Classifications of Parallel Computer Architectures and Programming Models

## 2.1 Flynn's Taxonomy of Parallel Computer Architectures

There are two types of information dealt by a processor, instruction and data [12]. Flynn's Taxonomy classifies Parallel Computer Architectures along these two independent dimensions of instruction and data. Depending on these dimensions, there are four types of computer architectures. These are Single Instruction Single Data, Single Instruction Multiple Data, Multiple Instruction Single Data and Multiple Instruction Multiple Data.

### Single Instruction Single Data (SISD)

Traditional single processor computer architectures are categorized as Single Instruction Single Data architecture. In this conventional architecture, only single stream of instructions can be executed at a time which operates on a single stream of data [13]. So if there is an instruction for increment operation, it will only increment a single variable at a time. This is how the traditional serial computers operate. Figure 2.1(a) shows such architecture.

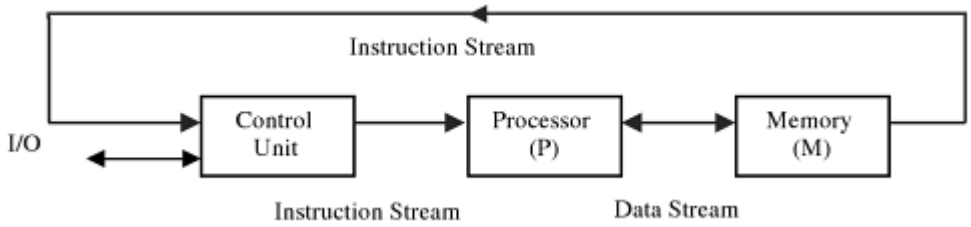


Figure 2.1(a): Single Instruction Single Data architecture [12]

In SISD architecture, there is one control unit, one processing unit and one memory. An instruction is fetched from memory and is passed to the control unit, which then passes this instruction to the processing unit. The processing unit analyzes the instruction, fetches the required data from memory and then executes the instruction on the fetched data. This whole process is done serially.

### Single Instruction Single Data (SIMD)

Single Instruction Multiple Data architecture executes same instruction on multiple data items. To execute instruction on multiple data simultaneously, SIMD architecture has more than one processing units. Hence SIMD architecture includes a single control unit, more than one processing units and a single memory unit. Such architecture is used to exploit data parallelism in an application, where processing on multiple data items are independent of each other. For example a *for loop* with independent iterations can be parallelized on SIMD architecture to speed up the application. All machines that have vector of instructions belong to SIMD architecture [13]. Figure 2.1(b) shows SIMD architecture.

### Multiple Instructions Single Data (MISD)

In Multiple Instruction Single Data architecture, a single stream of data flows to multiple processing units where each processing unit executes a different instruction on this data. There is no such architecture exists in practice [13].

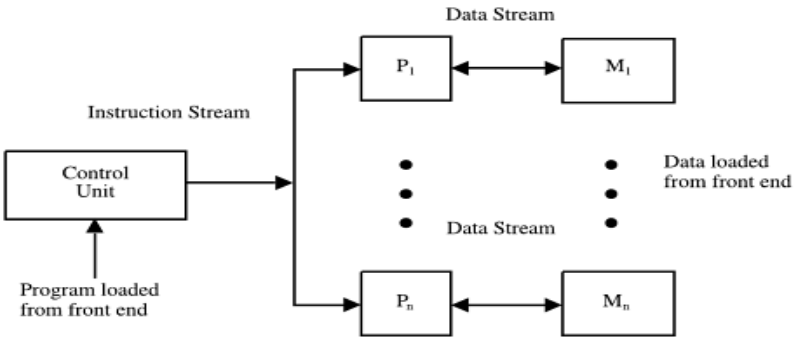


Figure 2.1(b): Single Instruction Multiple Data architecture [13]

### Multiple Instructions Multiple Data (MIMD)

Multiple Instructions Multiple Data architecture consists of many processors with each processor having its separate control unit. Thus MIMD architecture allows executing entirely different instructions on entirely different data simultaneously. MIMD can be used to exploit both data parallelism and task parallelism in applications, as it can leverage from separate control unit of each processor which can either execute same instruction or entirely different instruction on different data. Such task and data parallelism can either be synchronous or asynchronous [16]. Most of the current supercomputers fall into this category.

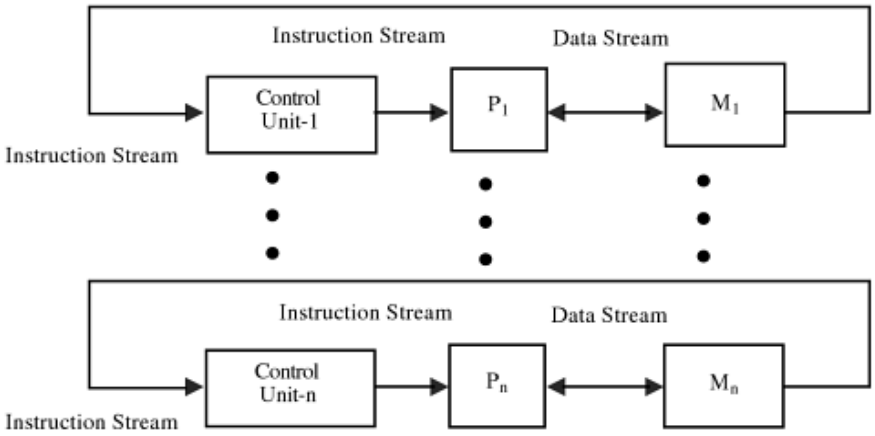


Figure 2.1(c): Multiple Instructions Multiple Data Architecture [13]

## 2.2 Parallel Computer Memory Architectures

There are three major parallel computer memory architectures used in parallel computing. They are named according to the way the data is passed from one processor to another for coordination among them. These are Shared Memory, Distributed Memory and Distributed-Shared Memory architectures.

### 2.2.1 Shared Memory Architecture

In shared memory architecture multiple processors have access to a single global memory and thus all processors sight a single common global address space [14]. In such memory architecture, coordination among different processors is done by reading and writing to a common memory location in global address space. However there are certain precautions which need to be taken into consideration for assuring that the correct value is read by a processor, as multiple processors can modify the value stored at a particular memory location. Figure 2.2(a) shows shared memory architecture.

The major advantage of such memory architecture is that coordination among different processors does not require explicit communication. Coordination among processors is as easy as using a single assignment statement in a program which modifies the value of data at particular memory location. Any change in value of data at some particular memory location is visible to all processors [14]. Hence to share information with other processor, once one of the processors has modified the data at particular memory location; other processors can simply read that memory location afterwards. However to assure that correct value is read by one processor, certain synchronization mechanism needs to be taken into account.

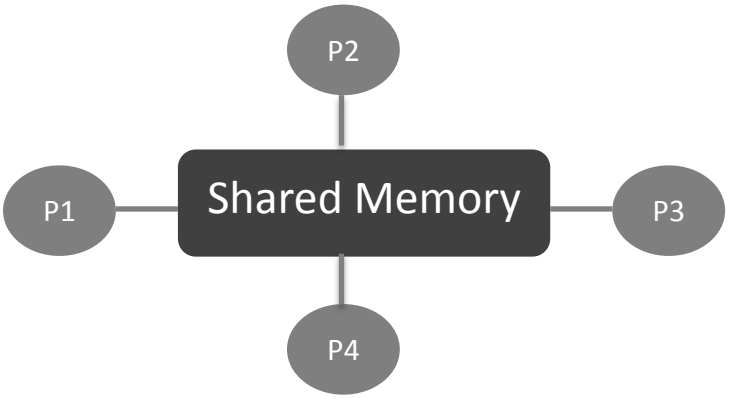


Figure 2.2(a): Shared Memory Architecture

There are also few performance bottlenecks of shared memory architecture. These include synchronization of data to be accessed by multiple processors (more specifically when data can be modified by more than one processor), high data traffic on path between processor and shared memory, and cache coherence. In shared memory architecture, each processor has its own local cache; this can result in each processor's local cache having different value of the data of a particular memory location [15]. There is need for cache coherence mechanism which can assure that correct value is written back to memory location before any other processors reads it. So that other processors read the correct value whenever they attempt to do so.

Shared memory architecture can be further divided into two categories depending on the time to access shared memory, namely Uniform Memory Access and Non-Uniform Memory access [16].

### 2.2.2 Distributed Memory Architecture

In distributed shared memory architecture each processor has its own local memory, thus the concept of global address space does not apply to distributed memory architecture [16]. In distributed programming environment, each processor with its own local memory is referred as node. Such a memory architecture requires an interconnect network for coordination among different nodes. As each node has its own local memory, any changes done to its local data is not visible to any other node unless a node explicitly coordinates this data with any other node. Hence, the issue of cache coherence does not appear in distributed memory architecture. As coordination among different nodes is done by absolutely sending or receiving messages in a program, this parallel computer architecture is also termed as Message Passing Multiprocessors. Figure 2.2(b) shows distributed memory architecture with communication between different nodes.

Though it seems like the need of explicit communication can result in performance bottleneck of distributed memory architecture, as now each node needs to coordinate its data (if required) to other node(s). But message passing is considered as irrefutable conqueror in terms of performance and scalability [1]. The most convenient feature of Distributed Memory architecture is that there is high scalability of memory with number of processors, unlike shared memory architecture where addition of more number of processors can result in high traffic on path between shared memory and

processors [16].

Addition of more processors can result in increasing probability of cache coherence issues in shared memory architecture, as now more processors will share the local memory. And therefore

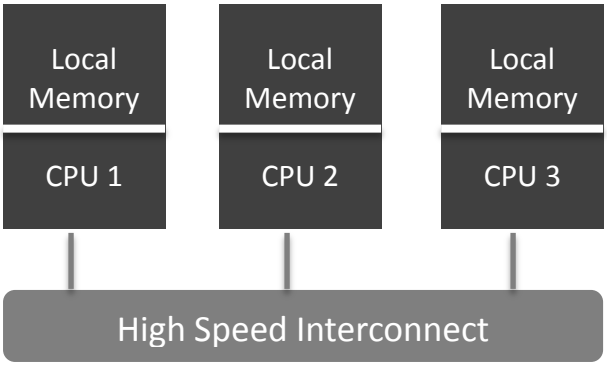


Figure 2.2(b): Distributed Memory Architecture

there are more chances of different cached value for each processor in shared memory system. Distribute Memory Architecture benefits from the fact that it does not face any overhead of maintaining a single value of data due to cache coherence issues. Hence Distributed memory architecture can scale without any such barrier of cache coherence.

### 2.2.3 Distributed-Shared Memory Architecture

Distributed-shared memory architecture acknowledges the advantages of both shared-memory and distributed memory architectures. DSM architecture provides an abstraction of shared memory model on top of distributed memory architecture, where each node can have multiprocessors communicating through shared memory [17]. It achieves such an abstraction by providing uniform address space for distributed system, such that the remote accesses are hidden from the programmer. Accessing a remote memory location in DSM architecture is as easy as in shared memory architecture, by simply using an assignment statement. Apart from leveraging from the ease of programming of shared memory architecture, it also enjoys the scalability of distributed memory architecture.

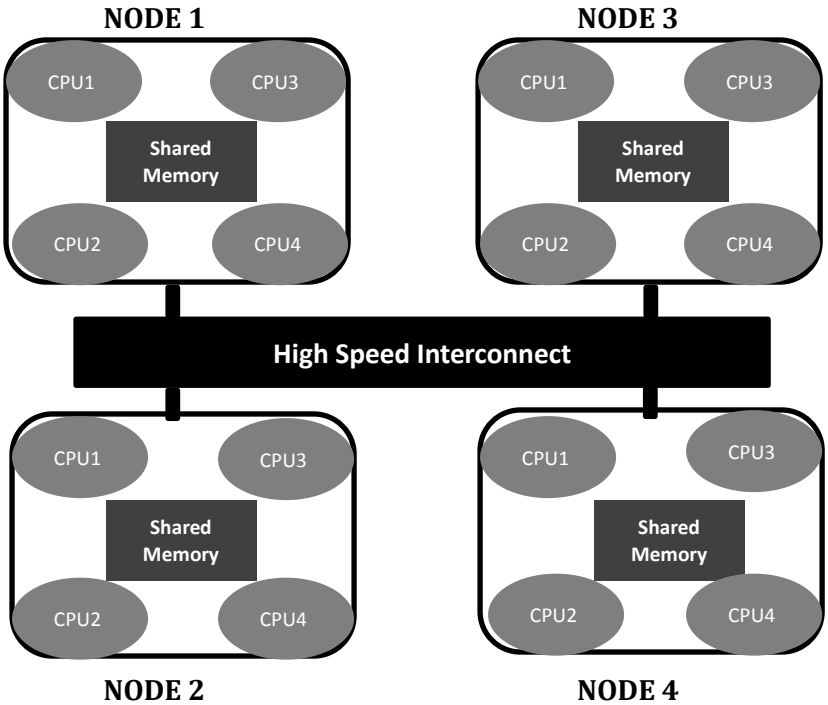


Figure 2.2(c): Distributed-Shared Memory Architecture

DSM architecture consists of multiple nodes connected by some high speed interconnection network. Each of these nodes has their own local memory and one or more processors (with each of them having its own local cache). The local memory of each node is mapped either fully or partially to global address space of distributed shared memory architecture [17], hence providing a uniform address space hiding local or remote access. It is then the responsibility of programmer to efficiently exploit the underlying shared (for processors on a same node) and distributed memories for communication bound and compute intensive problems respectively. A DSM also maintains a global table for keeping information about current state and location of each data [17], which helps in distinguishing between local and remote accesses.

Figure 2.2(c) shows distributed-shared memory architecture. There are total four nodes, each having four processors and a local shared memory. All nodes are interconnected using a high speed interconnect network. An abstraction of global shared memory is formed by mapping part of local shared memory of each node to global address space of DSM architecture.

## 2.3 Parallel Programming Models

There are several parallel programming models available currently. We have chosen three of these parallel programming models which are most commonly used and address parallel computer architectures discussed in section 2.2.

### 2.3.1 OpenMP

Parallel programming in shared memory architecture had always faced the scalability and portability issues, unlike message passing interface which provides good scalability as well as portability for applications on distributed memory architecture [14]. Though the development of Scalable Shared Memory Multiprocessors (SMPP) resolved the problem of scalability on shared memory architecture (with their scalable hardware support for cache coherence) [18], the issue of portability still confronted the use of shared memory parallel programming model. On the other hand, message passing interface lacks the ability to benefit from the shared memory architecture and also an application code is needed to be rewritten for MPI in order to divide the data structures and their processing among various nodes.

| **Serial Program** | **OpenMP Program** |
|---|---|
| void print_total_items (int items[], int array_length)<br>{<br>int total_items = 0;<br>int i;<br><br><br><br>for (i = 0; i < array_length; i++ )<br>  total_items += items[i];<br><br>printf (“Total Items: %d”, total_items);<br>} | void print_total_items (int items [], int<br>array_length){<br>int total_items = 0;<br>int i;<br><br>*#pragma omp parallel for shared(items, length)*<br>*private(i) reduction(+:total_items)*<br>for (i = 0; i < array_length; i++)<br>  total_items += items[i];<br><br>printf (“Total Items: %d”, total_items);<br>} |

Figure 2.3(a): Comparison of Serial and OpenMP code

OpenMP is an easy to use shared memory programming model, which resolves the issue of portability which other shared memory programming models lacked [18]. It is based on simple compiler directives, environment variable and library routines which make an application quite easy to parallelize. It parallelizes the application by simply using compiler directives and environment variables, which can be ignored by compiler and run-time system when the application is not compiled with OpenMP support. It is an industry standard API for shared memory parallel programming and has

support for C/C++ and FORTRAN. Portability is achieved by the use of compiler directives, as when the application is ported from multiprocessor to single processor environment, the compiler directives are simply ignored by the compiler and vice versa [14].

| Thread Creation | OpenMP Program |
|---|---|
| Only one thread before any Compiler directive is encountered.<br>----------<br>Creation of threads when a OMP Compiler directive is encountered. Number of iterations are divided among threads (array_length/4 per thread).<br>---------- | void print_total_items (int items [], int array_length){<br>int total_items = 0;<br>int i;<br><br>*#pragma omp parallel for shared(items, length) private(i) reduction(+:total_items)*<br>for (i = 0; i < array_length; i++)<br>total_items += items[i];<br><br>printf ("Total Items: %d", total_items);<br>} |

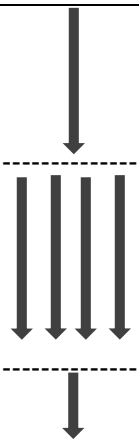

Figure 2.3(b): Thread Execution in OpenMP program of Figure 2.3(a)

Pthreads is another widely used shared memory parallel programming model but it is not intended for High Performance Computing applications, as it is based on task parallelism and has very little support for data parallelism unlike OpenMP which has great support for data parallelism [14].

Figure 2.3(a) shows a comparison of serial program with an OpenMP program, showing that very little effort is required to parallelize a serial code with OpenMP. The compiler directive in OpenMP version (starting with #pragma omp) informs the compiler that following for loop is to be parallelized. By default loop iterations are divided among number of processors available on machine, however a programmer can specify the number of threads using environment variable *OMP_NUM_THREADS*. Figure 2.3(b) represents the creation of threads in an OpenMP program shown in Figure 2.3(a).

### 2.3.2 Message Passing Interface

Message passing is an abstraction for parallel programming in distributed memory architecture. Message passing interface provides the standard library specifications for message passing programming. The data is coordinated between processors on different nodes by explicitly calling send and receive routines in a program. There are also certain extensions to conventional message passing model are provided in form of collective routines, parallel I/O, remote memory access and dynamic process creation [20]. MPI is not a message passing implementation itself; there are certain implementations of MPI like OpenMPI, MPICH, Intel MPI etc.

MPI provides a way in which different processors can communicate with each other. It does not depend on underlying memory architecture, meaning MPI can also be used on shared memory architecture as well as on distributed memory architecture. Of course on shared memory architecture, message passing can utilize shared memory for fast communication between processes [21].

MPI provides a rich set of library function calls. An MPI program needs to call these routines explicitly, there is no implicit parallelism provided by MPI, like OpenMP. A programmer has to divide the data structure and tasks among different processes and provide coordination of different processes explicitly using MPI library routines. Though it seems unsatisfactory to call MPI routines explicitly in a program and divide the work among processes manually, MPI provides very high scalability of Parallel applications [21]. Figure 2.3(c) shows a simple MPI program with each node printing its rank or node ID.

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv) {
  int rank;

  MPI_Init(&argc,&argv); /* Initialize the MPI environment */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); /*Assign a unique ID to each node*/

  printf("Hi from Node %d\n",rank); /*Each node prints its ID*/

  MPI_Finalize();
}
```

Figure 2.3(c): Simple MPI Program

### 2.3.3 Unified Parallel C

Unified Parallel C is a locality-aware distributed shared memory parallel programming extension of ANSI C and is based on Partitioned Global Address Space (PGAS). As the name suggest, unified parallel C provides an abstract view of unified global address space to a programmer in which remote access is very similar to a local access [1]. However access to an element in remote memory has an overhead as compared to a local memory access.

UPC achieves a global view of distributed shared memory by providing a source to source compiler. A source to source compiler translates a UPC code to a rudimentary C code with additional code for communication to access remote memories (which are hidden to user) [1]. The UPC-to-C translation allows the sequential C compiler to use available optimizations and produce an optimal machine language code [1].

UPC attracts HPC programmers due to its productivity. It has very simple syntax which makes it quite easy to program, like a single assignment statement can be used to access a remote memory location and domain decomposition can be achieved by simply replacing the predominant C language for loop with UPC version of for loop [4]. UPC has also very little set of extensions to C language, which are quite easy to remember, thus it gains the attention of comprehensive number of parallel programming users. UPC makes data distribution among threads straightforwardly using distributed arrays, which allow programmers to keep data close to the threads that will compute on them [4]. Hence, UPC is a very productive and easy to use parallel extension of C language. Figure 2.3(d) compares a serial version of C code with UPC code. It is quite reflective from this figure that with only few logical steps, UPC achieves parallelism in a distributed shared memory environment.

| Serial Program | UPC Program |
|---|---|
| void square_items (int items[], int array_length)<br>{<br>int i;<br><br>for (i = 0; i < array_length; i++ )<br>  items[i] *= items[i];<br>} | #include<upc.h><br><br>void square_items (shared int * items, int array_length){<br>int i;<br><br>**upc_forall** (i = 0; i < array_length; i++; **&items[i]**)<br>  items[i] *= items[i];<br>} |

Figure 2.3(d): Comparison of Serial C code with UPC code

In the next chapter the core concepts and features of Unified Parallel C are elaborated. This can help users in understanding our ported MD code to UPC.

# 3 Unified Parallel C

This chapter provides detailed explanation of parallel programming model Unified Parallel C. The major goal of this chapter is to introduce the users with ease of programming that UPC brings with its powerful extensions to ANSI C99 standard.

UPC is based on Partitioned Address Global Space and enjoys both the ease of programming in shared memory paradigm and scalability of distributed memory paradigm. UPC provides an abstraction of global address space which is divided equally among threads. To exploit the locality, UPC accommodates several constructs which allow placing data near the threads which compute on it [7][8]. Due to flat address space, any thread can access any memory location in shared address space irrespective of the fact that whether that memory location actually resides in local or remote memory. A user can access both local and remote memory locations in a similar manner, the low level communication to access remote memory location is hidden from users. However a user can explicitly check for affinity of a memory location to minimize the remote accesses.

UPC allows setting the number of parallel threads either at compile time with compiler flag –THREADS or at program start time by specifying argument to *upcrun* command. A UPC program runs in a SPMD fashion, where all threads executes the main function but can follow different execution paths to work on different data using available UPC constructs [5]. UPC threads run independently of each other, the only implied synchronization is at the beginning and termination of main function [5]. Hence it is the responsibility of the programmer to take care of necessary synchronization while accessing shared data by more than on threads. UPC is a derivative of distributed shared memory model. Apart from providing global shared address space, UPC also provides private address space for each thread which is only accessible by the thread who is owner of this private address space. This allows programmers to intelligently allocate the data in private and shared address spaces, such that only a data which has to be accessed excessively by one thread and rarely by other thread(s) needs to be allocated on region of shared address space (which has affinity to thread which computes intensively on this data) [5]. A data which remains local to a thread throughout the program execution should be allocated on private address space.

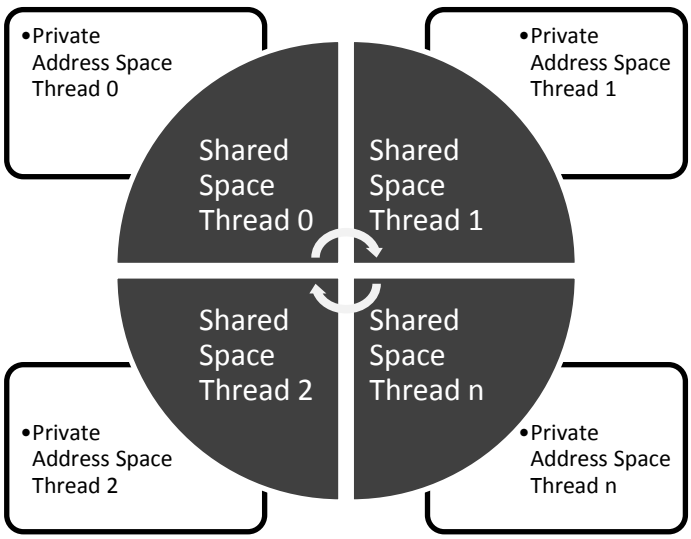


Figure 3: Memory Layout of UPC

Figure 4 shows memory layout provided by UPC. Each thread has its private address space which can only be accessed by thread who owns it, this is shown be rectangles around the circle in this figure. The circle in center shows a shared address space which is divided among all executing threads, each thread can access shared region which has affinity to it as well as shared region of any other thread. A

thread and data that has affinity to it are then placed on the same physical node by UPC run-time system. Hence, programmers can exploit locality by distributing the data cleverly among threads [1].

## 3.1 UPC Constants and Data Types

UPC provides constants to identify total number of threads at run-time and thread ID of each thread. The constant *THREAD* is a global constant which can be used at run-time in allocation of certain data structures and algorithms which requires knowledge of total number of threads. The global constant *THREAD* is visible to all threads. Whereas each thread has private copy of constant *MYTHREAD,* which stores unique thread ID for each thread. The constant MYTHREAD can be used to control the flow of execution of each thread, such that allowing each thread to follow a distinct execution path.

UPC provides extension to the ANSI C data types which apart from the type of the data also annotates where the data resides, private address space or shared address space. A memory can be allocated on shared address space by simply adding the prefix *shared* to the data type declaration. Whereas the traditional C data type declaration results in allocation of data in private address space [5]. Declaration of data as shared results in single copy of data which can be accessed by all threads, though this data would have affinity to certain thread(s). While conventional C data declaration will produce multiple copies of data, each thread having its own local copy of data residing in the private address space of thread. Figure 4.1(a) shows few examples of allocation of data in either private or shared address space in UPC.

| Data Declaration | Location of Data | Number of Copies of Data |
|---|---|---|
| int a; | Private Address Space | Equals to number of threads. Each thread has its own local copy of variable a. |
| *shared* int b; | Shared Address Space | One copy which resides in shared region having affinity to thread 0. |

Figure 3.1(a): Allocation of Data in Private or Shared Address Space

In Figure 3.1(a), the reason for variable *b* to have affinity to thread 0 is to conform the fact that first element of a *shared* array always has affinity to thread 0. Shared arrays are discussed later in section 3.2.

| Function/Structure Declaration | Restrictions |
|---|---|
| struct dataset {<br>int a;<br>shared int b;<br>int *c;<br>shared int *d;<br>} | Declaration of variable *b* is illegal as private structure cannot have a shared value. However a structure can contain a *private* pointer which points to some shared region. This is done in last statement where *d* is a private pointer to shared region. UPC pointers are discussed in detail later. |
| void illegalFuncDefinition() {<br>int a;<br>shared int b;<br>static shared int c;<br>shared int *d;<br>} | Declarations of *b* and *d* are illegal. |

Figure 3.1(b): Restrictions on shared variables declaration inside a block

Local variables of a function cannot be declared as shared due to the fact that local variables of a function have automatic storage duration. Local variables of a function have the lifetime limited to the duration of execution of the function call. After the function is returned, all its local variables are

destroyed and their storage is reoccupied automatically. To give some freedom, UPC accommodates declaration of *static shared* variables inside a function [5]. Static variables have their lifetime equal to the execution of complete program but their scope is limited to the function having its declaration. Furthermore, a private structure cannot contain any variable whose allocation is to be done on shared address space [5]. Figure 3.1(b) shows some restrictions on declaration of shared variables in a function definition and in a structure.

## 3.2 Distributed or Shared Arrays in UPC

UPC provides uncomplicated way of distributing data among multiple threads to aid exploitation of locality. This is achieved using shared arrays available in UPC. The elements of a shared array are distributed among threads in a round-robin fashion in chunks of consecutive elements enumerated by *BLOCK_SIZE* in the declaration of shared array. Below is the standard declaration of a shared array in UPC.

*shared* [*BLOCK_SIZE*] data_type array_name[array_length];

Consider an example in Figure 3.2(a) which shows distribution of array elements among three threads with a *BLOCK_SIZE* of 4.

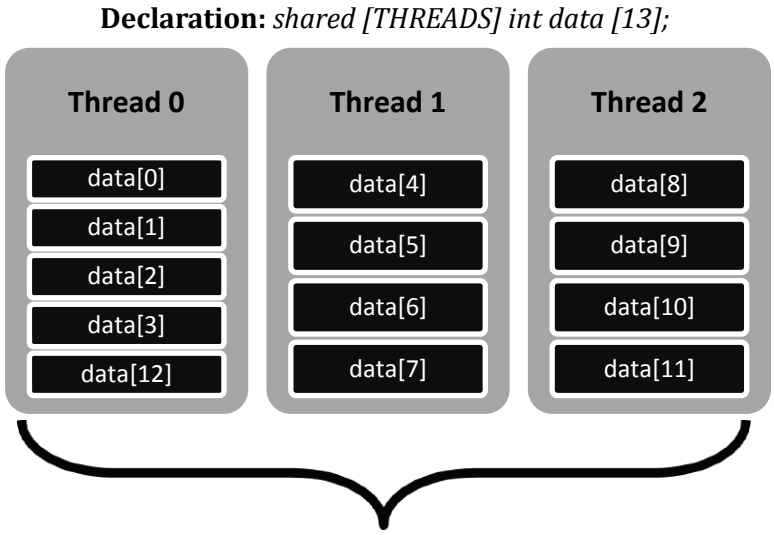


Figure 3.2(a): Distribution of shared array's elements

There are also few variants of shared array declaration. These are:

1) Allocates all elements of shared array in shared space of thread 0, using infinite *BLOCK_SIZE*.

   *shared* **[ ]** data_type array_name[array_length]; // using empty layout qualifier

   OR

   *shared* **[0]** data_type array_name[array_length]; // using *BLOCK_SIZE* equal to 0

2) Divides elements of shared array among threads with adjacent elements allocated in shared space of same thread. Thus it allows exploiting locality where frequent access to neighbor elements of shared array is required.

   *shared* **[*]** data_type array_name[array_length]; // using *BLOCK_SIZE* equal to *
3) Divides elements of shared array with default *BLOCK_SIZE* of 1.

*shared* data_type array_name[array_length]; // Not using any block qualifier

There are certain restrictions on declaration of shared arrays statically if the UPC program is not compiled with static number of threads (using flag -*THREADS*). Compiling a UPC program for pre-known number of threads aids the UPC compiler with the opportunity to optimize the UPC code. But it does not seem a good idea to compile a UPC program with the fix number of threads if the program is to be executed several times with each run-time environment containing different number of threads. The specification of total number of threads at run-time places some restrictions on the shared array declarations or distribution of data among threads using shared arrays [5]. In general, below are the restrictions if a UPC program is not compiled for static number of threads.

1) A shared array with **definite layout qualifier** must have length of **exactly** one of the dimensions of array equals to **scalar multiple** of total number of threads (use of static global constant THREADS in array dimension) [19]. Below are the examples obeying and violating this restriction.

   shared int data[2*THREADS]; // legal – (i)
   shared int data[5][THREADS]; // legal – (ii)
   shared int data[THREADS][5]; // legal – (iii)
   shared [2] int data[THREADS]; // legal – (iv)
   shared int data[20]; // illegal – (v)
   shared int data[THREADS][THREADS]; – (vi)
   shared int data[THREADS+4]; //illegal – (vii)

2) A shared array with **indefinite layout qualifier** cannot have any dimension of array depending on total number of threads (THREADS). Below are the examples obeying and violating this restriction.

   shared [] int data[4]; // legal – (viii)
   shared [] int data[2*THREADS]; //illegal – (ix)
   shared [] int data[THREADS+4]; //illegal – (x)

In general, when a UPC program is translated to dynamic threads environment, number of shared array elements to be distributed to each thread should be **constant** (i.e. varying number of threads should not result in different number of elements to be allocated to each thread). In above examples, statements (v), (vii) and (viii) are illegal because these statements allocate different number of elements to each thread when executed with different number of threads. For example, when statement (v) is executed with *THREADS* equals to 4, it allocates 5 elements to each thread. Whereas when the same statement (v) is executed with *THREADS* equals to 2, it allocates 10 elements to each thread. Thus elements per thread do not remain constant with varying number of threads, which is not allowed in UPC's dynamic threads environment.

## 3.3 UPC Pointers

UPC provides four distinct pointers classified according to the address space where these pointers reside and the address space which is pointed by these pointers. These four distinct pointer divisions are [5]:

i. Private Pointer directing towards Private Space
ii. Private Pointer directing towards Shared Space
iii. Shared Pointer directing towards Private Space
iv. Shared Pointer directing towards Shared Space

Figure 3.3(a) shows two pointer variants of UPC which have the property that they reside in one address space and point to other address space. These are Private Pointer to Shared Address Space

(PS) which resides in Private Address Space and Shared Pointer to Private Address Space (SP) that resides in Shared Address Space. The other two variants have the property that they reside in the same address space to which they point. These are Private Pointer to Private Address Space (PP) and Shared Pointer to Shared Address Space (SS).

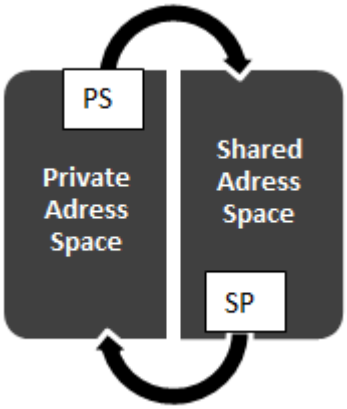


Figure 3.3(a): Location and target space of two UPC Pointer variants (PS and SP)

Figure 3.3(b) shows few examples of UPC pointer declarations to have a better practical understanding of these pointers. The use of *shared* keyword relative to the position of asterisk (*) in the UPC pointer declarations is important, as the position of asterisk distinguishes between different variants of UPC pointers. If the *shared* keyword appears after (or on the right side of) the asterisk in pointer declaration, then the declared pointer is shared and it has affinity to thread 0. Whereas if there is not any *shared* keyword after (or on the right side of) the asterisk in pointer declaration then the declared pointer is private and each thread has a local copy of this pointer. It is worthwhile to note here that the right side of asterisk identifies pointer type (either it is a private or shared pointer) while left side of asterisk identifies the type of pointer (integer, float or any other type) and the target address space (private or shared space). It is recommended not to use shared pointer to private space, as it allows to access private space of other thread, which obliterate the principal of private space of each thread visible only to itself [5].

| No. | Pointer Declaration Statement | Type of Pointer | Carries additional information than address of target memory location? |
|---|---|---|---|
| 1 | int * pointerA; | Private Pointer to Private Space | No |
| 2 | shared int * pointerB; | Private pointer to Shared Space | **Yes.** (i) Thread information, (ii) Virtual Address and (iii) Phase |
| 3 | int * shared pointerC; | Shared Pointer to Private Space | **Yes.** (i) Thread information, (ii) Virtual Address and (iii) Phase |
| 4 | shared int * shared pointerD; | Shared Pointer to Shared Space | **Yes**. (i) Thread information, (ii) Virtual Address and (iii) Phase |

Figure 3.3(b): UPC Pointer Examples

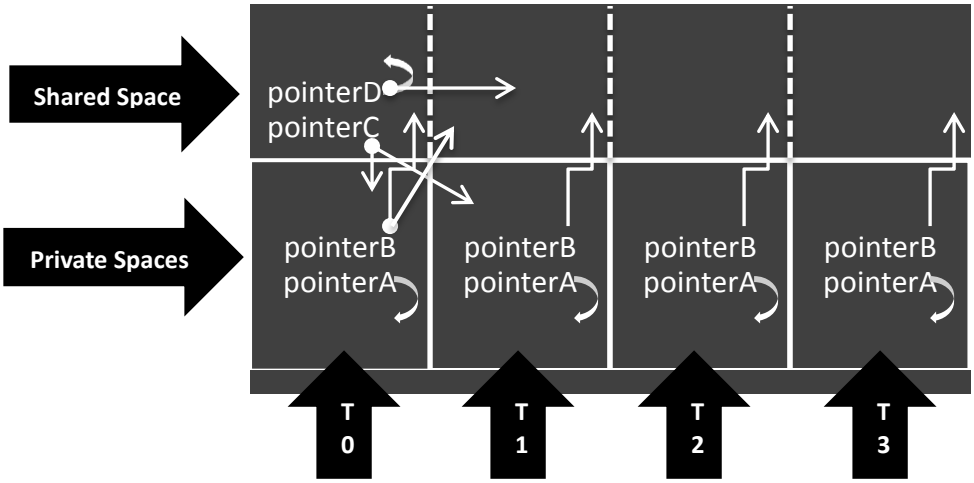


Figure 3.3(c): UPC Pointer Locations and their Target Address Space(s)

Figure 3.3(c) shows location of the pointers and their ability to point to one or both UPC address spaces for pointer examples of Figure 3.3(b). It should be clear from this figure that each thread has its own local copy of private pointers (pointerA and pointerB), whereas there is single copy each of shared pointers (pointerC and pointerD). As explained earlier, pointerA can only point to private space hence it doesn't contain any additional information about thread affinity, phase and virtual address of memory location it points to. Pointer pointerB is private pointer to shared address space and it can point to any memory location in shared space regardless of affinity of a memory location to any thread. Pointer pointerC is a shared pointer to private address space and hence it is discouraged to use it as it violates the fundamental basis of UPC address spaces. Pointer pointerD is a shared pointer to shared space. There is only one instance of pointerD which has affinity to thread 0 and it can point to any location in the shared space. Pointers to shared space (pointerB, pointerC and pointerD) store the additional information about thread affinity, virtual address and phase of memory locations they store.

Inherited from ANSI C, pointers to shared space can also be used to traverse shared arrays by advancing these pointers. Shared arrays in UPC comprise the blocking factor *BLOCK_SIZE* for distribution of consecutive elements to the same thread. Pointers to shared arrays are also required to be declared with blocking factor if they are advanced to go through elements of a blocked shared array. But one should be very careful while applying arithmetic to a pointer to shared array. To pass through the elements of a shared array correctly, the blocking factors of shared array and pointer to shared array must be equal; otherwise it can result in improper traversing of shared array elements [5]. Figure 3.3(d) shows an example program which highlights incorrect traversing of shared array due to dissimilar blocking factors of pointer and its target shared array. Figure 3.3(e) illustrates distributed array and its traversing for the program in Figure 3.3(d).

```
/* Traverse shared array using shared pointers – Program compiled for 3 threads */
#define BLOCK_SIZE 4
shared [BLOCK_SIZE] int array[14]; // declaration of shared array with blocking factor of 4
shared [BLOCK_SIZE] int * ps1; // declaration of pointer to shared space with blocking factor of 4
shared int * ps2; // declaration of pointer to shared space with default blocking factor of 1

ps1 = array; // Statement 1 -- ps1 point to starting address of shared array
ps1 = ps1 + 3; // Statement 2 -- results in correct traversing
ps2 = ps1 + 2; // Statement 3 -- results in correct traversing
ps3 = ps1 + 2; // Statement 4 -- results in incorrect traversing
```

Figure 3.3(d): Program for traversing a shared array using pointers

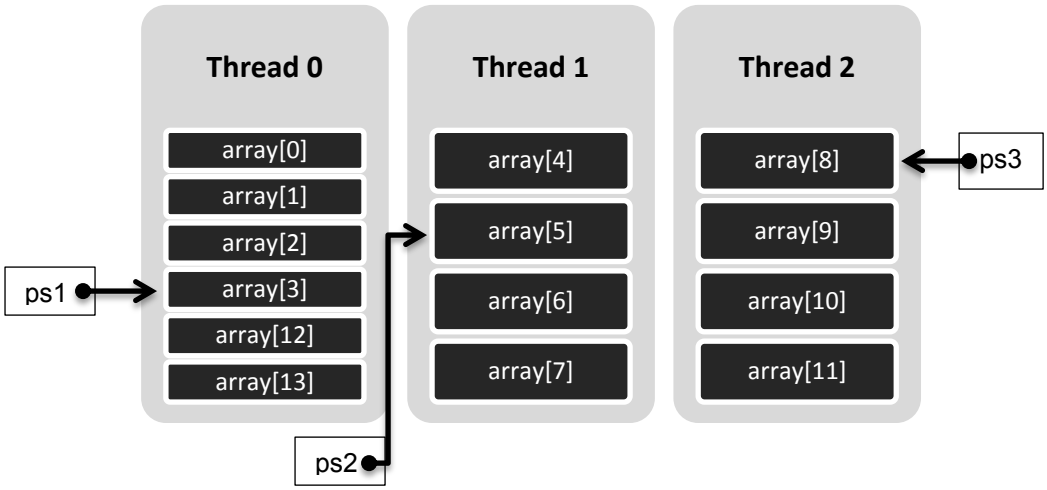


Figure 3.3(e): Illustration of traversing shared array for the program in Figure 3.3(d)

In figure 3.3(e), ps1 points to the fourth element of shared array as it was expected after execution of statements 1 and 2 in figure 3.3(d). Pointer ps2 points to sixth element of shared array as it was supposed to after the execution of statement 3 in figure 3.3(d). On the other hand, pointer ps3 points to ninth element of shared array, whereas it was expected to point to the sixth element like ps2. But due to default *BLOCK_SIZE* of 1 for ps3, it views the shared array as it has blocking factor of 1. Hence when incremented ps3 by 2 in statement 4 (of figure 3.3(d)), it traverses shared array in the pattern of fourth element to fifth and then eighth of shared array.

UPC permits casting among its pointer types [5]. This is useful as shared data which is local to a thread can also be accessed using a local pointer. Traversing of local shared data can also be done using arithmetic on local pointers in a normal way; this is so as local shared data is allocated adjacently in memory. Shared pointers carry additional information as compared to local pointers. This additional information is comprised of thread affinity, block address and phase (location of data within the block) of the data pointed by the shared pointer. Hence, casting a shared pointer to a local pointer will result in loss of this additional information that shared pointer carries. Whereas casting of a local pointer to shared pointer is not recommended [5] as it would give rise to wrong results. It is important to note here that when accessing a local shared data using local pointer, its programmer's responsibility to take care of lower and upper bounds of local shared data. Figure 3.3(f) shows an example of a program which shows how local pointers can be used to traverse local shared data.

```
/* Traverse local shared data using local pointers – Program compiled for 3 threads */
#define BLOCK_SIZE 4
shared [BLOCK_SIZE] int array[14]; // declaration of shared array with blocking factor of 4
shared [BLOCK_SIZE] int * ps; // declaration of pointer to shared space with blocking factor of 4
int * p; // declaration of local pointer

ps = &array[BLOCK_SIZE*MYTHREAD]; // ps point to starting address of local shared data at
                                  // each thread
p = (int *) ps; // casting of shared pointer to local pointer
p = p + 1; // points to second element of local shared data
p = p + 3; // crosses the upper bound of local shared data at thread 2 and thread 3 -- error
```

Figure 3.3(f): Example of traversing local shared data using local pointers

Figure 3.3(g) illustrates the error condition due to crossing the upper bound of local shared data when traversing local shared data using local pointers. Here thread 0 has enough number of elements that even after incrementing local pointer p in the last statement of figure 3.3(f), p still does not go beyond the upper limit of shared data elements of thread 0. However for threads 1 and 2, pointer p crosses the limit of total number of local shared data elements for these threads, which might result in erroneous results.

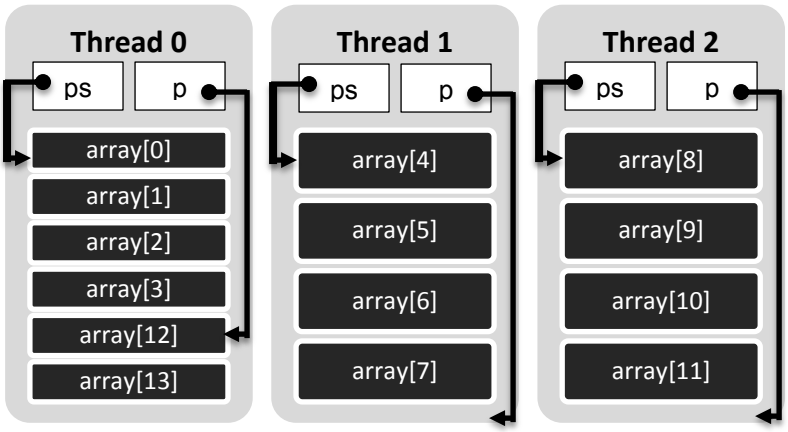


Figure 3.3(g): Traversing local shared data using local pointers for the program in Fig 3.3(f)

UPC facilitates with some very useful library routines which help in obtaining the information from the shared pointer that a shared pointer carries additionally as compared to a local pointer. This information can then be used to control the flow of execution of a program – different execution path for each thread, and also for manipulation of the shared data astutely. Among these library functions, the most commonly used are [5]:

**Function:** size_t upc_***thread***of(shared void * ps)
**Explanation:** This function returns the thread ID which has affinity to the shared data pointed by ps (pointer to shared space).

**Function:** size_t upc_***phase***of(shared void * ps)
**Explanation:** This function returns the phase of the shared data pointed by ps (pointer to shared space)..

**Function:** size_t upc_***addrfield***of(shared void * ps)
**Explanation:** This function returns the local address of the shared data pointed by ps (pointer to shared space).

## 3.4 Domain Decomposition in UPC

Domain decomposition in parallel programming involves dividing the data in chunks [16], such that each thread or process may then work on the chunk of data it has been compelled to compute on. UPC programs executes in SPMD fashion [1], where each thread computes on a portion of the total data. To divide the job among threads, it is necessary that a thread can identify itself and all other threads in the environment which are working on the same job [5]. This ability of each thread to recognize itself and other threads allows efficiently controlling the flow of execution for each thread and associating each thread with a particular chunk of data to compute on.

UPC is very powerful extension of C, which provides variable declaration semantics that associate a data with particular thread by providing affinity of the data to some thread. Knowing the affinity of the data, a programmer can then cleverly divide the job among threads by compelling each thread to compute on data that each thread has the affinity to. This in turn also reduces the remote memory accesses significantly in distributed memory architecture as UPC compilers map a thread and its associated data to the same physical node [5]. UPC also provides local and global constants, *MYTHREAD* and *THREADS* respectively, which empowers each thread with the ability of identifying itself and other threads in the UPC run-time environment. These constants help a programmer to manage the workload among the threads.

Figure 3.4(a) shows an example program for basic domain decomposition and workload division among threads using UPC variable declaration semantics and constants, *MYTHREAD* and *THREADS*. This program calculates the sum of squares two vectors of length *THREADS* each in a parallel fashion, where each thread calculates the sum of the elements of these vectors which has affinity to this thread. Statements 1,2 and 3 in program decompose the domain and assign each thread one element of each of vectors v1,v2 and result to compute on. Statement 4 in program uses the local constant of each thread, *MYTHREAD*, to compute on data which has affinity to it. Finally thread 0 prints the sum of the squares of two vectors, v1 and v2. Figure 3.4(b) visualizes the example presented in Figure 3.4(a).

```
#include <upc.h>

shared int v1[THREADS]; // statement 1
shared int v2[THREADS]; // statement 2
shared int result[THREADS]; // statement 3

int main (void) {
        /* Initialized shared arrays*/
        result[MYTHREAD] = v1[MYTHREAD] * v1[MYTHREAD] +
                                v2[MYTHREAD]*v2[MYTHREAD];  // statement 4

        upc_barrier; // wait for other threads to finish

        if(0 == MYTHREAD)
                for(int i = 0; i < THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.4(a): Example of basic domain decomposition and work division

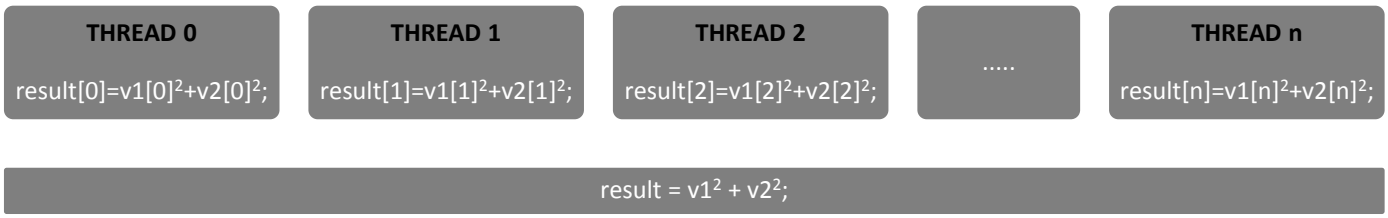


Figure 3.4(b): Visualization of example in Figure 4.4(a)

In addition to variable declaration semantics and constants that allow efficient domain decomposition and workload division among threads, UPC provides a dominant *upc_forall* work sharing iteration statement which is a UPC version of ANSI C for loop. It allows to distribute the loop iterations among threads, treating each loop iteration as an independent unit of job [5]. However this is a programmer's responsibility to make sure that there is no data dependency among the loop iterations, so they can be executed in parallel without producing wrong results.
The *upc_forall* statement is quite similar to the conventional *for loop* of C language. It has first three arguments similar to the ANSI C for loop, whereas it has one additional argument that decides which thread is responsible for the execution of a particular iteration. Given below is the syntax of *upc_forall* statement:

**upc_forall(initialization_expression; loop_condition; increment_expression; affinity)**

The fourth argument, affinity, can either be a pointer to shared address space or an integer value. In case the fourth argument is a pointer to shared space, a particular iteration is executed by the thread which has affinity to the shared data pointed by the pointer used as fourth argument. When the fourth

argument is an integer value, a particular iteration is executed by the thread having the value of its local constant *MYTHREAD* equals to the 'given integer value modulo *THREADS* (global constant)'. Each thread can execute more than one iteration; however each iteration must only be executed by one particular thread [5]. Consider an example in figure 3.4(c), which is bit modified version of example in figure 3.4(a). Here each thread calculates the sum of the squares of three elements of vectors v1 and v2. The work distribution here is achieved using *upc_forall* statement with its fourth parameter being a pointer to shared region. Each thread executes three iterations of the *upc_forall* statement, whereas each iteration is executed by only one thread.

```
#include <upc.h>

shared int v1[3*THREADS]; // statement 1
shared int v2[3*THREADS]; // statement 2
shared int result[3*THREADS]; // statement 3

int main (void) {
        /* Initialized shared arrays*/
        upc_forall(int i = 0, i < 3*THREADS; i++; &result[i])
                result[i] = v1[i]*v1[i] + v2[i]*v2[i];  // statement 4

        upc_barrier; // wait for other threads to finish

        if(0 == MYTHREAD)
                for(int i = 0; i < 3*THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.4(c): Work division using *upc_forall* statement with affinity argument of shared pointer

Figure 3.4(d) shows the example presented in Figure 3.4(c) but with fourth argument of *upc_forall* statement being an integer value. Here ith iteration is executed by a thread whose value *of MYTHREAD* equals *i%THREADS*. This results in round robin distribution of iterations among UPC threads.

```
#include <upc.h>

shared int v1[3*THREADS]; // statement 1
shared int v2[3*THREADS]; // statement 2
shared int result[3*THREADS]; // statement 3

int main (void) {
        /* Initialized shared arrays*/
        upc_forall(int i = 0, i < 3*THREADS; i++; i)
                result[i] = v1[i]*v1[i] + v2[i]*v2[i];  // statement 4

        upc_barrier; // wait for other threads to finish

        if(0 == MYTHREAD)
                for(int i = 0; i < 3*THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.4(d): Work division using *upc_forall* statement with affinity argument of integer value

Apart from having an integer value or pointer to shared as a fourth argument of *upc_forall* statement, the fourth argument of *upc_forall* statement can also be a *continue* statement or an empty field [19]. In case the fourth argument is a *continue* statement or an empty field, all iterations of *upc_forall* statement are executed by all threads. UPC also allows nesting of *upc_forall* statements. Nevertheless in nested *upc_forall* statements, the outermost *upc_forall* statement determines the work distribution among threads [19], whereas the inner *upc_forall* statements act as having their fourth argument of affinity being a continue statement or an empty field. This results in all iterations of inner *upc_forall* statements in nested hierarchy to be executed by all threads, such that they are not divided among threads.

## 3.5 Dynamic Shared Memory Allocation

There are certain scenarios in programs when the amount of memory required for data is not known before the run-time. Such requests for capturing the memory are fulfilled by allocating memory on the heap at run-time, known as dynamic memory allocation. Dynamic memory allocation also provides the advantage of reusing the allocated memory during the course of a program. It is especially very useful when large memory is acquired during the run-time [5], this large chunk of memory can be released when it is not required by the program anymore and later can be used for further memory allocation requests.

Dynamic memory allocation in the private address space of UPC is achieved by using the inherited ANSI C routines, e.g. malloc(). However for dynamic memory allocation in shared address space, UPC provides exclusive functions [8]. There are two major function variants of dynamic shared memory allocation in UPC, collective and non-collective. A collective function for dynamic shared memory allocation needs to be called by all threads for the allocation of required shared memory, whereas a non-collective function for dynamic shared memory allocation is only required to be called by a single thread to acquire the required memory on shared address space [5]. UPC provides three functions for allocating the memory dynamically on its shared address space, while it provides only a single function for releasing the memory allocated on shared space [8][19]. The functions for dynamic shared memory allocation are *upc_all_alloc*, *upc_global_alloc* and *upc_alloc*. Each of these functions covers a vast detail and needs to be discussed separately.

***upc_all_alloc():***

*upc_all_alloc* is a collective function call for dynamic memory allocation on shared space and hence it must be called by all threads in a program with the same argument values. Below is the function definition of *upc_all_alloc*:

shared void * upc_all_alloc(size_t NUM_BLOCKS, size_t BLOCK_SIZE)

This function returns a single pointer to allocated memory in the shared space, which is suitably aligned and can be assigned to any type of pointer to shared [19]. The allocated shared memory is distributed among threads in a round robin fashion in the chunks of *BLOCK_SIZE*. The *BLOCK_SIZE* is determined using second argument of this routine, which mentions total number of bytes in each block. Whereas the total number of blocks which sum up to total shared memory is determined using first argument of this routine. The total number of bytes allocated in shared memory is therefore *NUM_BLOCKS* times *BLOCK_SIZE*.

The shared memory allocated dynamically with the above mentioned upc_all_alloc function is similar to shared array allocated statically in following way [5]:

shared [BLOCK_SIZE] char [BLOCK_SIZE * *NUM_BLOCKS*];

If the memory is allocated successfully using *upc_all_alloc* function, an identical pointer value is returned at all threads. However if memory could not be allocated, a NULL pointer to shared space is

returned [5]. The memory allocated using *upc_all_alloc* function needs to be deallocated explicitly if not required by the program anymore.

Figure 3.5(a) shows an example of dynamic shared memory allocation using function *upc_all_alloc*. Statement 1 declares a private pointer to shared space 'ps' with the blocking factor of 2, thus each thread has a private copy of this pointer. Statement 2 allocates memory dynamically on the shared space and returns a pointer pointing at the start of allocated memory. Hence, ps of each thread points to the start of allocated shared memory. This can be seen in Figure 3.5(b) which illustrates the example in Figure 3.5(a) when executed with four threads. With statement 3, each thread initializes its local shared portion of the allocated shared memory. Finally, using statement 4 each thread prints the values of elements of shared array which have affinity to it.

```
#include <upc.h>
#define BS 2
int main (void) {

        shared [BS] int * ps; // statement 1

        ps = (shared [BS] int *) upc_all_alloc(THREADS, BS * sizeof(int)); // statement 2

        upc_forall(int i = 0, i < BS*THREADS; i++; &ps[i])
                ps[i] = MYTHREAD;  // statement 3

        upc_forall(int i = 0, i < BS*THREADS; i++; &ps[i])
                printf("THREAD[%d] ps[%d] = %d", MYTHREAD, i, ps[i]);  // statement 4

        return 0;
}
```

Figure 3.5(a): Example program dynamic shared memory allocation using *upc_all_alloc*

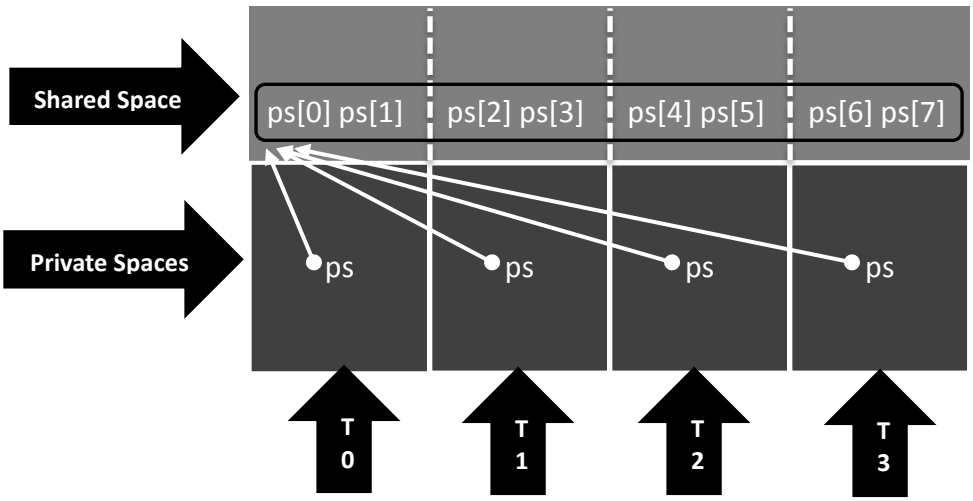


Figure 3.5(b): Dynamic memory allocation for the example in Figure 3.5(a) executed with 4 threads

***upc_global_alloc():***

*upc_global_alloc* is a non-collective function call for dynamic memory allocation on shared space and hence each thread can call this routine independently of any other thread and with different arguments compared to other threads. Below is the function definition of *upc_global_alloc*:

shared void * upc_global_alloc(size_t NUM_BLOCKS, size_t BLOCK_SIZE)

If the memory is allocated successfully on shared space, a pointer to shared space which points to allocated memory is returned, otherwise a NULL pointer to shared space is returned. The allocated shared memory is distributed among threads in a round robin fashion in the chunks of *BLOCK_SIZE*. The *BLOCK_SIZE* is determined using second argument of this routine, which mentions total number of bytes in each block. Whereas the total number of blocks which sum up to total shared memory is determined using first argument of this routine. The total number of bytes allocated in shared memory is therefore *NUM_BLOCKS* times *BLOCK_SIZE*.

It is critical to highlight the major difference between *upc_all_alloc* and *upc_global_alloc* routines. If *upc_global_alloc* is called by several threads, each of these threads gets a different memory allocated in shared address space and each thread gets a pointer to shared which points to the start of memory allocated for it. Whereas a call to *upc_all_alloc* results in a single memory space allocated in shared address space and all threads get identical pointers which point to the start of this allocated memory. The shared memory allocated dynamically by calling *upc_global_alloc* function is similar to the shared array allocated statically in following way [5]:

shared [BLOCK_SIZE] char [BLOCK_SIZE * *NUM_BLOCKS*];

```
#include <upc.h>
#define BS 2
int main (void) {

        shared [BS] int * ps; // statement 1

        ps = (shared [BS] int *) upc_global_alloc(THREADS, BS * sizeof(int)); // statement 2

        upc_forall(int i = 0, i < BS*THREADS; i++; MYTHREAD)
                ps[i] = MYTHREAD;  // statement 3

        upc_forall(int i = 0, i < BS*THREADS; i++; MYTHREAD)
                printf("THREAD[%d] ps[%d] = %d", MYTHREAD, i, ps[i]);  // statement 4

        return 0;
}
```

Figure 3.5(c): Example program dynamic shared memory allocation using *upc_global_alloc*

Figure 3.5(c) shows an example of dynamic shared memory allocation using function *upc_global_alloc*. Statement 1 declares a private pointer to shared space 'ps' with the blocking factor of 2, thus each thread has a private copy of this pointer. Statement 2 results in the allocation of separate dynamic memory on shared space for each thread. Each thread gets a pointer to shared which points at the start of memory allocated to it. This can be seen in Figure 3.5(d) which illustrates the example in Figure 3.5(c) when executed with four threads. With statement 3, each thread alone initializes its allocated shared memory completely. Finally, using statement 4 each thread prints the values of elements of shared array allocated in result of its call to *upc_global_alloc*.

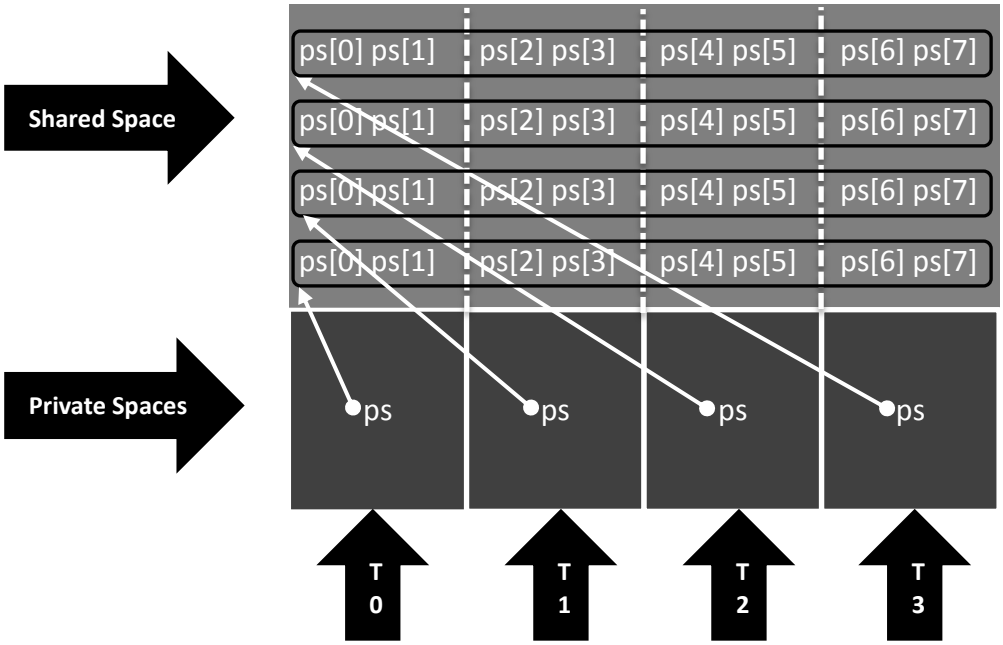


Figure 3.5(d): Dynamic memory allocation for the example in Figure 3.5(c) executed with 4 threads

***upc_alloc():***

UPC also facilitates with the ability for each thread to dynamically allocate memory on its local shared portion of global shared address space. This can be quite convenient in certain scenarios in a program where each thread has to work alone on some data and it is unaware of size of this data before run-time. So each thread can dynamically allocate memory on its local portion of shared space and hence effectively exploit the locality. This suits the intent if the shared data is not to be accessed by a thread other than the thread that has this data in its local portion of shared space, so there will only be local accesses to this data. Nevertheless this data can also be accessed by other threads if desired. Below is the function definition of *upc_alloc*:

shared void * upc_alloc(size_t BLOCK_SIZE)

If the memory is allocated successfully on local shared portion of calling thread, a pointer to shared space which points to the allocated memory is returned, otherwise a NULL pointer to shared space is returned. As the function takes only one argument which is *BLOCK_SIZE*, the allocated memory has the size of *BLOCK_SIZE* bytes and it has affinity to thread which called *upc_alloc* function.

Figure 3.5(e) shows an example of dynamic shared memory allocation using function *upc_alloc*. Statement 1 declares a private pointer to shared space 'ps' with indefinite blocking factor, thus each thread has a private copy of this pointer. Statement 2 results in the dynamic memory allocation on local shared portion of the thread which calls the function *upc_alloc*. Each thread gets a pointer to shared which points at the start of memory allocated on its local shared portion of the shared address space. This can be seen in Figure 3.5(f) which illustrates the example in Figure 3.5(e) when executed with four threads. With statement 3, each thread initializes its allocated shared memory. Finally, using statement 4 each thread prints the values of elements of shared array allocated in result of its call to *upc_alloc*.

```
#include <upc.h>
#define BS 2
int main (void) {

        shared [] int * ps; // statement 1

        ps = (shared [] int *) upc_alloc(BS * sizeof(int)); // statement 2

        upc_forall(int i = 0, i < BS; i++; MYTHREAD)
                ps[i] = MYTHREAD;  // statement 3

        upc_forall(int i = 0, i < BS; i++; MYTHREAD)
                printf("THREAD[%d] ps[%d] = %d", MYTHREAD, i, ps[i]);  // statement 4

        return 0;
}
```

Figure 3.5(e): Example program dynamic shared memory allocation using *upc_alloc*

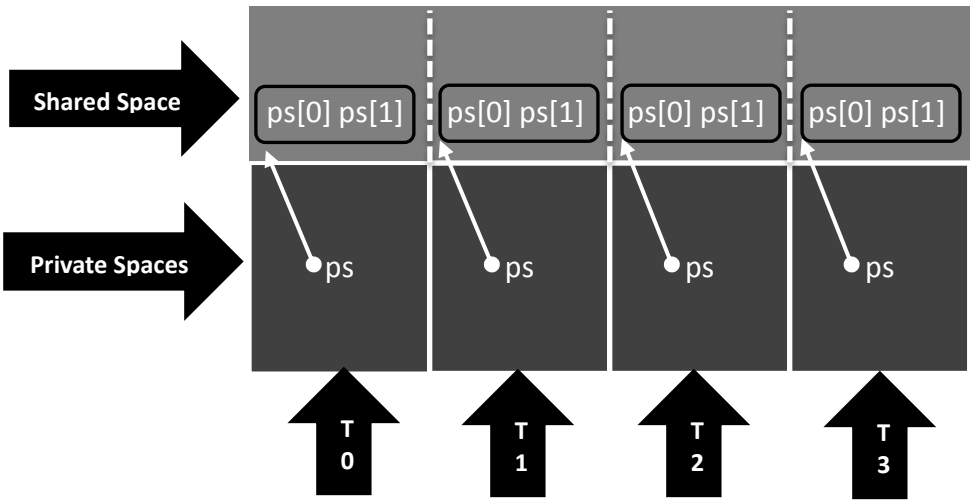


Figure 3.5(f): Dynamic memory allocation for the example in Figure 3.5(e) executed with 4 threads

UPC provides only a single function for dynamic shared memory deallocation. *upc_free* releases the memory acquired by any of the routines *upc_all_alloc*, *upc_global_alloc* or *upc_ alloc*.

***upc_free():***

*upc_free* releases the dynamically allocated memory on shared address space. This is quite useful in applications which work on large data sets; dynamically allocated memory which is not required anymore by the program can be released and made available for future allocation requests. UPC retains the ability of ANSI C to reuse a dynamically allocated memory by providing *upc_free* function which allows releasing dynamically allocated memory on the shared space. The memory allocated dynamically on the private space can be released by intrinsic ANSI C function *free()*. Below is the function definition of *upc_free*:

void upc_free(shared void * ps)

*upc_free* takes only one argument, which is pointer to shared space that points to the dynamically allocated memory on shared space that is to be released. If the pointer passed to this function points to the memory which has already been released or it does not point to any of the storages allocated dynamically on shared space, the behavior of this call is undefined [19]. Figure 3.5(g) shows an

example of releasing a dynamically allocated shared memory. This is the same program as in Figure 3.5(a) but it releases the allocated memory (at statement 5) once it is not required by program anymore. This can be seen in Figure 3.5(h) which illustrates the example in Figure 3.5(g) after the execution of statement 5.

```
#include <upc.h>
#define BS 2
int main (void) {

        shared [BS] int * ps; // statement 1

        ps = (shared [BS] int *) upc_all_alloc(THREADS, BS * sizeof(int)); // statement 2

        upc_forall(int i = 0, i < BS*THREADS; i++; &ps[i])
                ps[i] = MYTHREAD;  // statement 3

        upc_forall(int i = 0, i < BS*THREADS; i++; &ps[i])
                printf("THREAD[%d] ps[%d] = %d", MYTHREAD, i, ps[i]);  // statement 4

        upc_barrier; // wait fot other threads

        upc_free(ps); // statement 5

        return 0;
}
```

Figure 3.5(g): Example program for releasing dynamically allocated shared memory using *upc_free*

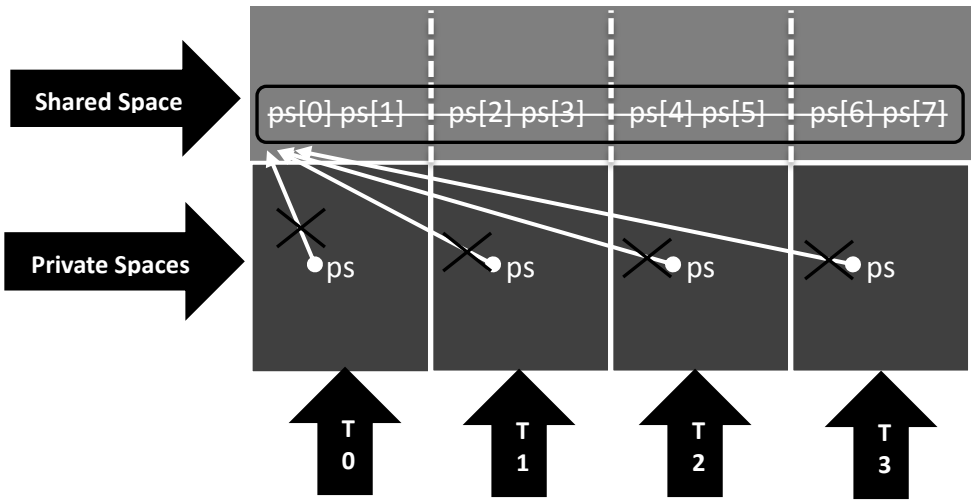


Figure 3.5(h): Illustration of example in Figure 3.5(g) after execution of statement 5

## 3.6 Synchronization and Memory Consistency

Synchronization is the fundamental and one of the most troublesome issues in parallel programming [23]. In our most of daily routines we need to synchronize for our certain actions, for example synchronizing for the time to use a shower when there is only one shower and there is more than one person who needs to use the shower [23]. Similarly in parallel programs when a data is to be modified by more than on thread or process, it requires appropriate synchronization among threads or processes to avoid corruption of the data. To be more precise, synchronization in parallel programs guarantees to produce the same result as if the program is executed serially [5], such that the part of the total work that each thread does is coordinated accurately with the other threads. However one

should not use excessive synchronization as it would result in high execution time of the application and the overall goal to speed up the application will be affected. UPC categorizes its synchronization constructs into two significant classes, control and data [5].

Control synchronization deals with allowing a thread to progress, such that whether a thread should proceed executing after a certain point in program or wait there for other threads to finish their job. This defines a scenario when the data modified by one thread can be used later by some other thread. Similarly whether a thread should execute a region in code atomically or not comes under the division of control synchronization. UPC accommodates with two control synchronization constructs, barrier synchronization and locks, which addresses above issues.

Data Synchronization deals with the order of modification of the data by different threads and the instance at which a data should become visible to other threads after being modified by a thread [5]. UPC also accommodates with data synchronization constructs; these are fences and memory consistency qualifiers.

### Barrier Synchronization

Barrier synchronization is the simplest synchronization mechanism in parallel programming. Barrier synchronization enables multiple threads to coordinate their arrival at a particular point in program during its execution [24]. No thread can proceed further unless all threads arrive at this point of execution. When all threads reach at this point, only then any thread can proceed further. Barrier synchronization can be used in scenarios when the data being modified by one of the threads can later be used by one or multiple auxiliary threads. It confirms that all the data has been modified which can be used by other threads, so all threads can now proceed in doing their tasks. Barrier synchronization is commodious in a situation where one phase of program must be completed by all threads before the next phase starts [24].

UPC provides two variants of barrier synchronization, blocking and non-blocking (or split-phase) barrier synchronization. In blocking barrier synchronization, a thread cannot proceed executing after arriving at particular synchronization point in the program unless all threads arrive at that point. Following is the syntax for blocking barrier synchronization [5]:

upc_barrier <expression>;

The <expression> in semantic of upc_barrier is optional and is an integer expression. This integer expression can be used to tag different synchronization phases of program with distinct integer values, providing ease of debugging when an error occurs during a particular phase in the program. It also enables to activate one barrier per group of threads where different group of threads calls *upc_barrier* with different integer values, and hence also makes it easy to debug the program in such cases. Figure 3.6(a) shows an example of using blocking barriers. This example is the modified version of example in Figure 3.4(d). Here thread 0 initializes the two vectors and later each thread calculates the sum of squares of the elements of the shared array which has affinity to this thread. Hence all threads must wait for the initialization of the vectors v1 and v2, which is achieved using *upc_barrier* call in program at statement 4. Similarly thread 0 also waits for other threads for completion of their part of computation at statement 5 before printing the result. Figure 3.6(b) illustrates the behavior of each thread after executing the *upc_barrier* call at statement 4 in program of Figure 3.6(a).

In non-blocking or split-phase barrier synchronization, a thread can proceed executing after arriving at particular synchronization point in the program and wait for other threads at later point in program. Hence it divides a synchronization phase in two synchronization points. At first it notifies all threads after arriving at particular point in program, and then proceeds with some computation which does not affect other threads. At the second synchronization point, this thread waits for other threads and cannot proceed unless all other threads have reached the first synchronization point. This enables overlapping of computation and communication among threads, unlike blocking barrier

```
#include <upc.h>

shared int v1[3*THREADS]; // statement 1
shared int v2[3*THREADS]; // statement 2
shared int result[3*THREADS]; // statement 3

int main (void) {
        if(MYTHREAD == 0)
                for (int i = 0; i < 3*THREADS; i++) {
                        v1[i] = i;
                        v2[i] = i;
                }

        upc_barrier; // statement 4
        upc_forall(int i = 0, i < 3*THREADS; i++; i)
                result[i] = v1[i]*v1[i] + v2[i]*v2[i];  // statement 5

        upc_barrier; // statement 6

        if(0 == MYTHREAD)
                for(int i = 0; i < 3*THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.6(a): Example program using blocking barrier

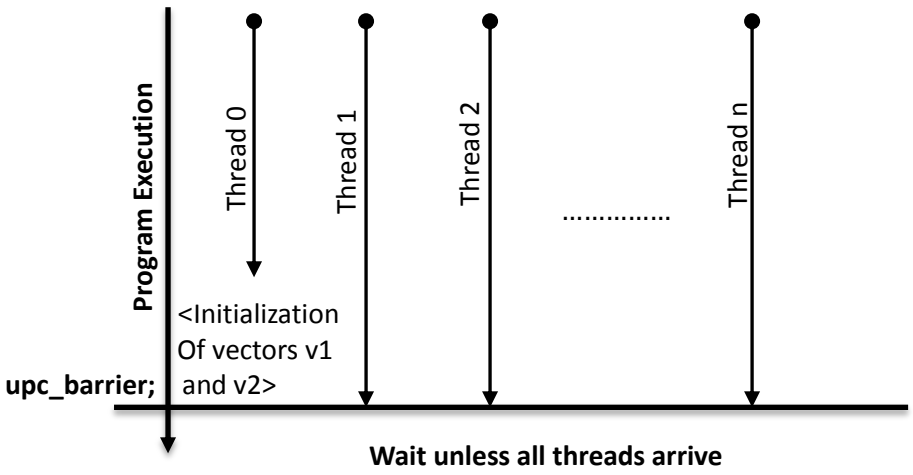


Figure 3.6(b): Behavior of each thread after executing the *upc_barrier* call

synchronization where all threads must wait for other threads after reaching a single synchronization point. Therefore it saves machine cycles from being wasted by not waiting for other threads, instead it can utilize those machine cycles for its local computation or the computation that does not affect any other thread. However it must wait for other threads before starting next part of computation that affects other threads. Following is the syntax for non-blocking or split phase barrier synchronization [5]:

```
upc_notify <expression>;
upc_wait<expression>;
```

When a thread reaches at first synchronization point, it calls *upc_notify* to inform other threads about its arrival at that synchronization point. This thread then can continue doing its local computation

before entering the next phase of computation that affects other threads. Before starting next phase, this thread calls *upc_wait* which assures that all other threads have completed the current phase, such that all threads have called *upc_notify* for current phase. The <expression> in both upc_notify and upc_wait statements serves the same purpose as it does in blocking barrier synchronization. Nevertheless in a same phase, both *upc_notify* and *upc_wait* should use same integer value (if used) to match each other for one particular phase, otherwise a run-time error is generated [5].

```
#include <upc.h>
shared int v1[3*THREADS]; // statement 1
shared int v2[3*THREADS]; // statement 2
shared int result[3*THREADS]; // statement 3

int main (void) {
        if(MYTHREAD == 0)
                for (int i = 0; i < 3*THREADS; i++) {
                        v1[i] = i;
                        v2[i] = i;
                }

        upc_notify; // statement 4

        /*Local Computation*/
        if(MYTHREAD != 0)
                printf("THREAD[%d] is doing local computation\n", MYTHREAD);

        upc_wait; // statement 5

        upc_forall(int i = 0, i < 3*THREADS; i++; i)
                result[i] = v1[i]*v1[i] + v2[i]*v2[i];  // statement 6

        upc_barrier; // statement 7
        if(0 == MYTHREAD)
                for(int i = 0; i < 3*THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.6(c): Example program using non-blocking barrier

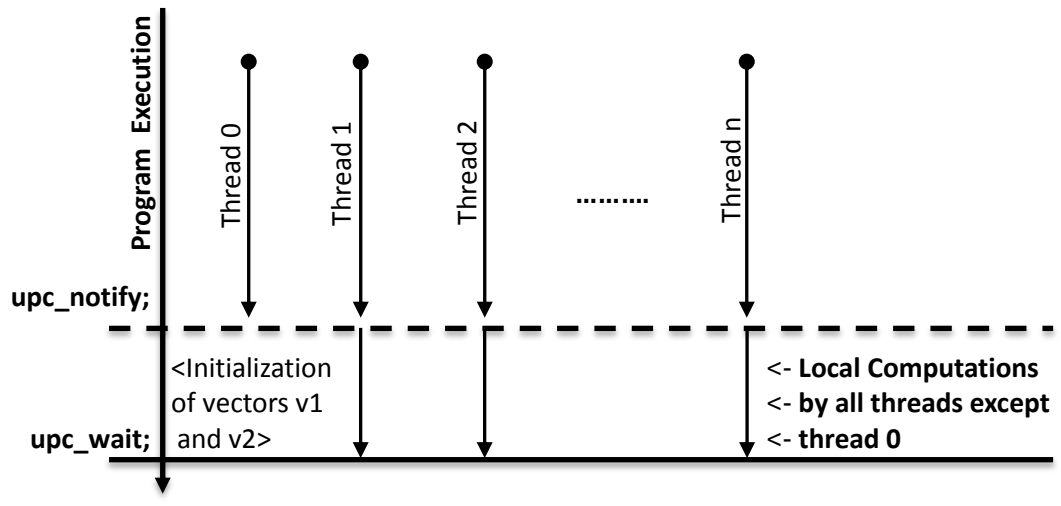


Figure 3.6(d): Behavior of each thread after executing the *upc_notify* and *upc_wait* calls

Figure 3.6(c) shows an example of using non-blocking barriers. This example is the modified version of example in Figure 3.6(a). Here all threads call *upc_notify* after arriving at first synchronization point at statement 4 and then performs their local computation. Before each thread starts calculating sum of squares of the elements of the shared array which have affinity to it, it calls *upc_wait* so it can be assured that thread 0 has initialized the vectors v1 and v2. Figure 3.6(d) illustrates the behavior of each thread after executing the *upc_notify* and *upc_wait* calls at statements 4 and 5 respectively in program of Figure 3.6(c).

The nesting of non-blocking barriers is not allowed, hence for each *upc_notify* call, the next call should be to its respective *upc_wait* call (i.e. with the same integer expression or no integer expression). Two successive *upc_notify* calls are not permitted. It is important to highlight the fact here that a call to *upc_barrier* function is similar to calling *upc_notify* and *upc_wait* functions back-to-back in the program [5]. Therefore using a sequence of calls *upc_notify*, *upc_barrier* and *upc_wait* is also not allowed as it leads to nesting of non-blocking barrier calls as well.

### Locks

Locks are used to permit access to the critical section of the program in a one-thread at a time manner. While one thread is inside the critical section, no other thread can enter the critical section. Lock is a data structure which has two states, locked and unlocked [5], which are changed atomically. Hence, allowing only a single thread to modify the state of the lock without any race condition. When a thread needs to access critical section, which must be executed by only single thread at a time, it must acquire lock for the critical section. And when a thread has acquired a lock, the other threads trying to access the critical section are not able to acquire the lock unless the occupier thread of the lock releases it exclusively at the end of critical section.

In UPC, a lock can be declared as a variable of type *upc_lock_t* [19]. The type *upc_lock_t* is non-transparent as it varies for different implementations. Hence a UPC lock can only be accessed using a pointer to lock [5]. A variable declared of type *upc_lock_t* is a private pointer-to-shared lock object. However using functions *upc_phaseof*, *upc_threadof* and *upc_addrfield* of on the variable declared as *upc_lock_t* results in undefined behavior so they should be avoided [19]. Locks are allocated by threads dynamically either in a collective or non-collective manner. The collective function for allocating a lock is:

upc_lock_t * upc_**all**_lock_alloc();

The function *upc_all_lock_alloc* takes no argument and returns a pointer-to-shared to the allocated lock. All threads in the UPC environment must call this routine for the allocation of the lock, and as a result a pointer-to-shared is returned to each thread which points to a same single lock allocated by calling this routine. The initial state of a lock is *unlocked* which can be changed by a separate routine discussed later, *upc_lock*.

The non-collective function for allocating a lock is:

upc_lock_t * upc_**global**_lock_alloc();

The function *upc_global_lock_alloc* takes no argument and returns a pointer-to-shared to the allocated lock. Each thread which calls this function gets a different instance of lock. Hence when called by all threads or more than on thread in a program, each thread gets a different pointer-to-shared to a distinct lock instance.

A lock allocated dynamically either collectively or non-collectively by a UPC thread can be deallocated when it is not required in program anymore. The routine for deallocation of a lock is:

void upc_lock_free (upc_lock_t * ps_lock);

The function *upc_lock_free* takes only one argument which is pointer-to-shared to a lock. If the passed pointer does not point to any lock which is allocated in program by either *upc_all_lock_alloc* and *upc_global_lock_alloc* functions, the behavior of the function is undefined [19]. And if the passed pointer is a NULL pointer, nothing happens as a result to the call *upc_lock_free* [19]. It is important to mention that if a lock is allocated in a collective manner, only a single thread should deallocate this lock. The routine for deallocating a lock is non-collective and hence if called by many threads to release the same lock, it would result in undefined behavior.

UPC provides two functions to change the state of a lock from unlocked to locked or to acquire a lock, blocking and non-blocking. The blocking function for acquiring a lock is:

void upc_lock (upc_lock_t * ps_lock);

The function *upc_lock* takes only one argument which is pointer-to-shared to a lock. It changes the state of the lock from unlocked to locked, pointed by the passed pointer-to-shared of type *upc_lock_t*. If the lock is already occupied by some other thread in UPC environment, a thread making call to routine *upc_lock* is blocked or cannot proceed further in program unless the lock is released by the thread who occupies it by changing its state from locked to unlocked [5][19]. Nevertheless if the call to *upc_lock* is made by a thread to acquire a lock which is already occupied by this thread, the behavior is undefined [19]. The blocking nature of function *upc_lock* results in wastage of machine cycles when a lock is not available to be locked [5]. The non-blocking function to change the state of the lock from unlocked to locked is:

int upc_lock_attempt (upc_lock_t * ps_lock);

The function *upc_lock_attempt* takes only one argument which is pointer-to-shared to a lock. It returns an integer value of either 1 or 0 depending upon if the lock is acquired or not. Like *upc_lock*, it also changes the state of the lock from unlocked to locked, pointed by the passed pointer-to-shared of type *upc_lock_t*. However if the lock is already in locked state (or occupied) by some other thread, a thread making call to routine *upc_lock_attempt* is never blocked and gets the return value of 0. And if the lock is available to be locked, a thread making call to routine *upc_lock_attempt* changes state of the lock from unlocked to locked and gets the return value of 1. The non-blocking nature of function *upc_lock_attempt* makes it possible to overlap the process of acquiring a lock and some computation to utilize the machine cycles efficiently unlike the function *upc_lock* [5]. When a thread already occupies a lock and calls the function *upc_lock_attempt*, the behavior is not known [19].

UPC facilitates with the function *upc_unlock* to release the lock or to change the state of a lock from locked to unlocked. This same function can be used to release a lock occupied by either a call to *upc_lock* or *upc_lock_attempt*. The definition of the function *upc_unlock* is:

void upc_unlock (upc_lock_t * ps_lock);

The function *upc_unlock* takes only one argument which is pointer-to-shared to a lock. It changes the state of the lock to unlocked from locked, pointed by the passed pointer-to-shared of type *upc_lock_t*. If the lock is not occupied by a thread who calls the function *upc_unlock*, the behavior is undefined. Similarly if a lock is already in the unlocked state, the behavior of a call to function *upc_unlock* is undefined.

Figure 3.6(e) shows the program of Figure 3.6(a) with some additions to calculate the sum of all components of result vector. Thread 0 initializes the two vectors and later each thread calculates the sum of squares of the elements of the shared array which has affinity to this thread. Each thread needs to access the critical region which is protected by using a lock, declared in statement 2 and allocated collectively by all threads in statement 3. Inside critical region, a thread modifies a shared variable global_sum (declared in statement 1) which is used to calculate the sum of all components of result vector (Statement 5). Once a thread has completed its access to critical region, it releases the lock (in

statement 6) so other threads can acquire the lock to access critical region.

```
#include <upc.h>

shared int v1[3*THREADS];
shared int v2[3*THREADS];
shared int result[3*THREADS];
shared int global_sum = 0; // statement 1
upc_lock_t * sum_lock; // statement 2

int main (void) {
        sum_lock = upc_all_lock_alloc(); //statement 3

        if(MYTHREAD == 0)
                for (int i = 0; i < 3*THREADS; i++) {
                        v1[i] = i;
                        v2[i] = i;
                }

        upc_barrier;

        upc_forall(int i = 0, i < 3*THREADS; i++; i)
                result[i] = v1[i]*v1[i] + v2[i]*v2[i];

        upc_lock(sum_lock); // statement 4

        /*Critical Region*/
        global_sum += result[MYTHREAD]; // statement5

        upc_unlock(sum_lock); // statement 6

        upc_barrier;

        if(0 == MYTHREAD)
                for(int i = 0; i < 3*THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.6(e): Use of lock function

## Memory Consistency Modes and Fence

UPC provides two constructs for data synchronization, which deal with the order of modification of the data by different threads and the time at which a data should become visible to other threads after being modified by a thread [5]. These constructs are memory consistency mode and fence.

Memory consistency mode is provided either to allow or disallow the compiler to reorder the memory references. Compilers might use its inherent optimization capabilities to reorder the memory references in a program which do not affect the final result of a program and result in an optimized code [5]. UPC provides two memory consistency modes, relaxed and strict. The default memory consistency mode is relaxed. The relaxed consistency mode allows the compiler to reorder the memory references to produce an optimized code. Strict memory consistency mode prevents compiler to reorder the memory references and also enables preceding modifications to data visible to all threads.

Memory consistency modes can be applied at various levels in program, a particular variable or array, a section of program, or the complete program. To apply a strict memory consistency mode to a variable, the variable MUST be declared as shared and with the strict reference-qualifier [5][19], as shown below:

**strict** *shared* double sum;

If a strict reference-qualifier is not used with shared variable declaration, the default memory consistency mode is used for accesses to such variable which is relaxed. UPC provides pragmas to apply a particular memory consistency mode to a section of program. The syntax of memory consistency pragma is:

#pragma upc [relaxed | restrict]

If the pragma appears outside a section of the code (enclosed by curly brackets), the memory consistency mode specified in pragma applies till the use of next pragma or till the end of program if no pragma is used later in program to specify a particular memory consistency mode. And if the pragma is used inside a section of code, the memory consistency mode specified in pragma applies only to that section of code [19]. This is shown below. The left side shows the application of relaxed memory consistency mode only to a section whereas right side shows its application unless next pragma is encountered in the program or till the end of the program.

```
{ // Section begins                    #pragma upc relaxed
#pragma upc relaxed        OR         { // Section begins
.                                     .
.                                     .
} // Section ends                     } // Section ends
```

For a memory consistency mode to be effective in complete program, UPC provides two header files, whose inclusion identifies a particular memory consistency in the complete program. These header files are *upc_relaxed.h* and *upc_strict.h*. In scenario when different memory consistency modes are applied at different levels in program, the priority of memory consistency mode at a particular level prevails. The application of memory consistency mode at variable level has highest priority, then the memory consistency mode at section level has mid-priority and the memory consistency mode at program level has the lowest priority [5][19].

Fence is another powerful construct for memory consistency provided by UPC. It asserts that all the references to shared objects are completed before the use of fence statement in program, *upc_fence*. Therefore all the changes made to shared data before the fence statement are visible to all threads in UPC environment after the call to *upc_fence* returns. Hence it also makes sure that no access to a shared data which appears after *upc_fence* statement in program is done before the call to *upc_fence* statement [5], i.e. providing restrictions for compiler optimizations to reorder the statements.

## 3.7 UPC Collectives

There are certain requirements in programs where multiple threads need to coordinate their work with each other to have global view of the total work done. UPC provides collective library for such requirements in programs, which allows to exploit data locality effectively i.e. each thread computes on portion of the total data which has affinity to it and later all threads coordinate their work with other threads to have a final computed value on complete data. The standard header file for UPC collective library is *upc_collective.h*.

A UPC collective function is completed in three phases, which are action, notification and wait. In action phase, all threads compute the result depending upon the operation specified in a collective call.

A thread enters a notification phase once it has completed the computation it was responsible for and hence it notifies other threads in system about it. Finally a thread enters a wait phase, it which a thread can either wait for receiving notifications from other threads about completion their work or can proceed further in program. UPC provides the power to control synchronization in these phases by providing a flag argument of type *upc_flag_t* in all UPC collective functions [19]. UPC provides constants which can be used to form the value to be passed as argument of type *upc_flag_t* in collective functions. These constants are of two types, IN and OUT. The IN constants control when a UPC collective function can start reading or writing the shared data once it is called. Similarly, the OUT constants control when a call to UPC collective function should return. The syntax of IN constants is described below:

Syntax: UPC_*IN*_**X**SYNC
where **X** can be NO, MY or ALL as explained below.

*NO* – The collective function can start reading or writing the shared data when the first thread has entered the collective call.
*MY* – The collective function can only read or write the shared data that has affinity to threads which have already called the collective function.
*ALL* – The collective function can only start reading or writing the shared data once all threads in UPC environment have called the collective function. It assures that all threads read the same data as if any thread is still manipulating shared data which is to be used by other threads, the collective function does not begin reading or writing the shared data [19].

The syntax of OUT constants is described below:

Syntax: UPC_*OUT*_**X**SYNC
where **X** can be NO, MY or ALL as explained below.

*NO* – The collective function can read or write shared data unless atleast one thread is not returned from collective call.
*MY* – The collective function can return in a thread when all read and writes to shared data which has affinity to this thread has been completed.
*ALL* – The collective function cannot return unless all reads and writes of shared data irrespective of affinity to any thread have been completed. This asserts that once a thread returns from the collective call, it will not read any earlier value of output shared data [19]. However it does not imply barrier synchronization at the end of the collective call. Hence if a single thread enters the collective call and is able to begin read or write data without waiting for other threads, it is possible that this single thread returns from the collective function if no other thread has yet called the collective function [19]. Therefore for cases where it is required to make sure that all threads return from the collective call before proceeding further in the program, an explicit barrier synchronization statement is required in program.

There are two major types of UPC collective functions based on the functionality of operation specified in UPC collective calls. These are Relocalization and Computation collective functions [5]. The relocalization collective functions deal with the movement of the data between threads whereas computation collective functions deal with the application of various mathematical and logical operations on data distributed among threads.

In our work we have used computation collectives, so we will only discus one or two computation collective routines in our report. First we need to explain computation operations used in these collective functions. The computation operations have type *upc_op_t* and some of these are described below [19]:

UPC_ADD: Addition of shared data distributed among threads
UPC_MULT: Multiplication of shared data distributed among threads

UPC_AND: Bitwise AND operation of shared data distributed among threads (undefined behavior for floating values)
UPC_OR: Bitwise OR operation of shared data distributed among threads (undefined behavior for floating values)
UPC_XOR: Bitwise XOR operation of shared data distributed among threads (undefined behavior for floating values)
UPC_LOGAND: Bitwise Logical AND operation of shared data distributed among threads
UPC_LOGOR: Bitwise Logical OR operation of shared data distributed among threads
UPC_MAX: Operation to find maximum value among shared data distributed among threads
UPC_MIN: Operation to find minimum value among shared data distributed among threads

### Computation Reduction Operations

The most commonly used computation collective function is a Reduction function. A reduction function simply applies the operation specified in one of its arguments to the distributed shared data among threads. The syntax of a reduction collective function is [19]:

void upc_all_reduce_**T**(shared void * destination, shared const void * source, upc_op_t operation, size_t numElements, size_t BLOCK_SIZE, TYPE (*func) (TYPE,TYPE), upc_flag_t synMode)

The bold letter T in the definition of reduction function is replaced by a particular string which represents a specific data type of source and destination data. The letter T in function definition can have following values [19][5]:

| Value of T | Corresponding data type | Value of T | Corresponding data type |
|---|---|---|---|
| C | signed char | L | signed long |
| UC | unsigned char | UL | unsigned long |
| S | signed short | F | float |
| US | unsigned short | D | double |
| I | integer | LD | long double |
| UI | unsigned integer | | |

The arguments of reduction function are explained below:

1. shared void * destination – private pointer to shared destination buffer
2. shared const void * source – private pointer to shared source buffer
3. upc_op_t operation – reduction operation (UPC_ADD etc.)
4. size_t numElements – Number of elements in source buffer
5. size_t BLOCK_SIZE – Blocking factor of source buffer
6. TYPE (*func) (TYPE,TYPE) – NULL
7. upc_flag_t synMode – Synchronization mode controlled by passing a constant formed by OR-ing UPC_*IN*_**X**SYNC and UPC_*OUT*_**X**SYNC constants

When a call to the reduction function returns, the value in destination buffer is the result of operation (specified in third argument of function) applied to all elements of shared source array. For example, for a UPC_ADD operation the value in destination buffer after reduction function returns is:

destination = source[0] + source[1] + …… + source[numElements-1]

If the *BLOCK_SIZE* specified in fifth argument of reduction operation has a value greater than 0, the reduction operation view the source pointer as it points to a shared memory having total elements equal to numElements and blocking factor of *BLOCK_SIZE* [5]. For example, for a value of T being I, it view the source shared array pointed by source pointer as following static shared array declaration:

shared [BLOCK_SIZE] int [numElements]

However if the *BLOCK_SIZE* specified in fifth argument of reduction operation has a value of 0, the reduction operation view the source pointer as it points to a shared memory having total elements equal to numElements and blocking factor equals to indefinite layout qualifier. It means that all elements of the shared source array have affinity to a single thread, i.e. the elements of source shared array are not distributed among threads. For example, for a value of T being I, it view the source shared array pointed by source pointer as following static shared array declaration:

shared [] int [numElements]

```
#include <upc.h>

shared int v1[3*THREADS];
shared int v2[3*THREADS];
shared int result[3*THREADS];
shared int global_sum = 0; // statement 1

int main (void) {

        if(MYTHREAD == 0)
                for (int i = 0; i < 3*THREADS; i++) {
                        v1[i] = i;
                        v2[i] = i;
                }

        upc_barrier;
        upc_forall(int i = 0, i < 3*THREADS; i++; i)
                result[i] = v1[i]*v1[i] + v2[i]*v2[i];

        upc_barrier;
        upc_all_reduceI( &global_sum, &result, UPC_ADD, 3*THREADS, 1, NULL, UPC_IN_NOSYNC |
                        UPC_OUT_NOSYNC ); // statement2
        upc_barrier;

        if(0 == MYTHREAD)
                for(int i = 0; i < 3*THREADS; i++)
                        printf("Sum of ith element=%d\n", result[i]);
        return 0;
}
```

Figure 3.7: Use of reduction function

Figure 3.7 shows the program of Figure 3.6(e) with a modification that it now utilizes *upc_all_reduceI* function to calculate the sum of all components of result vector instead of calculating it using critical section surrounded by a lock.

The other computation reduction function which is quite similar to the *upc_all_reduceT* function is *upc_all_prefix_reduceT*. The only different in the later from former is that the destination buffer has the same layout as the source buffer and along with final the intermediate results are also saved in elements of destination buffer in following way:

$$\text{destination[i]} = \sum_{k=0}^{i} \text{source[i]}$$

The other computation reduction function *upc_all_sort* which sorts a shared array using a user-defined function specified in one of its arguments.

## 3.8 Shared Data Movement Functions in UPC

UPC inherits the ANSI C string handling function *memcpy* for the movement of data when both source and destination buffers are in private address space of a thread. However for the movement of data when atleast one of the source or destination buffers in located in shared memory, UPC facilitates with string handling functions for shared data. These are *upc_memcpy*, *upc_memget* and *upc_memput*.

The *upc_memcpy* function is used to copy data when both source and destination buffers are allocated in shared memory. The syntax of *upc_memcpy* function is:

void upc_memcpy(shared void * destination, shared const void * source, size_t numElements);

It view source and destination buffers similar to the following static shared array declaration:

shared [] char array[numElements];

Hence it is employed to move shared data which has affinity to one thread to same or other thread in UPC environment [19].

The *upc_memget* function is used to copy data when the source buffer is allocated in shared space and destination buffer is allocated in private space of a thread who calls this function. The syntax of *upc_memcpy* function is:

void upc_memget(void * destination, shared const void * source, size_t numElements);

It views source buffer similar to the following static shared array declaration:

shared [] char array[numElements];

whereas it views the destination buffer as conventional ANSI C declared array, as shown below:

char array[numElements];

The *upc_memput* function is used to copy data when the source buffer is allocated in private space of a thread who calls this function and destination buffer is allocated in shared space. The syntax of *upc_memput* function is:

void upc_memput(shared void * destination, const void * source, size_t numElements);

It views the source buffer as conventional ANSI C declared array, as shown below:

char array[numElements];

whereas it views destination buffer similar to the following static shared array declaration:

shared [] char array[numElements];

In the next chapter we will explain that how we have ported out CMD code to UPC which involves design of new algorithms and data structures under UPC utilizing the features of UPC discussed in this chapter. We will also highlight performance tuning and optimization techniques that we implemented for developing our algorithms.

## 3.9 Performance Tuning and Optimization of UPC Programs

There are various general techniques to improve the performance of parallel programs which include efficient use of inter-process communication, synchronization techniques, techniques for balancing work load among threads and techniques to avoid redundant computations. However even after utilizing these techniques efficiently, there are some important aspects to consider which can further enhance the performance of a UPC program. These include UPC compiler optimizations, UPC run-time system's proactive behavior, and manual optimizations to be done in UPC programs by the programmers [5].

A UPC compiler translates a UPC code to ANSI C code independent of particular system architecture, which is then compiled with the ANSI C compiler and linked to the UPC run-time system. UPC run-time system is responsible for creation of threads, distribution of shared data among threads and execution of code in parallel fashion [2][5]. The ANSI C compiler is free to optimize the translated C code to achieve efficient execution of a UPC program. However it is the responsibility of UPC-to-C translator to pass sufficient information to the backend ANSI C compiler for optimizations. The UPC run-time system can examine the shared data accesses and perform communication optimizations [2]. UPC programmers can employ manual optimization techniques to aid compiler and UPC run-time system optimizations. UPC compilers are still in processes of integrating many useful optimization techniques in it; therefore manual optimizations are required for achieving better performance of a UPC program [5].

The most instrumental manual optimization that can aid both UPC-to-C translator and UPC run-time system is to explicitly distinguish a local and remote shared data access. UPC provides construct which can be utilized for investigating the locality of a shared data and hence enables to explicitly express a remote or local shared data access. The manual optimization techniques are discussed below:

**Design of locality aware algorithms**
Designing of locality aware algorithms can result in huge performance boost of a UPC application. This can be achieved by architecting an algorithm in such a way that each thread majorly computes on local portion of the distributed data allocated in shared address space. This results in very less or no access to remote shared data which has an access overhead compare to the local shared data access. Hence locality aware algorithms allow UPC programmers to distribute the total data among all threads and then each thread works on the portion of data which has affinity to it [5]. We have developed locality aware algorithms for most of the functions in our CMD code.

**Pointer Optimizations**

Pointer optimization is a technique to access local portion of shared space by a thread using local pointer instead of point-to-shared. This can be done by checking for the affinity of data pointed by shared pointer and casting this shared pointer to local pointer if the data lies in the local portion of shared space of a thread. Accessing a memory using shared pointer involves overhead as compared to accessing it using local pointer.

**Efficient Use of UPC locks and Barrier Synchronization**

Locks are used to allow access to the critical section of the program in a mutually exclusive manner. While on thread has acquired a lock for a particular section of code, no other thread can execute that section of code unless the thread which occupies the lock releases the lock explicitly. Synchronization barriers are used when the next section of program cannot be started unless the previous section of program has been completed by all threads. Such that each thread waits after the completion of one section and can only continue proceeding further in program once all threads have completed the previous section. It is very necessary to use these synchronization constructs very carefully in program, while inefficient use of locks can lead to deadlock among threads, the barrier synchronization results in excessive synchronization as it results in providing barrier to the progression of each thread.

Hence it is necessary to use these constructs effectively to avoid excessive synchronization as well as deadlock among threads. UPC also provides a very useful non-blocking variant of barrier synchronization, called split-phase barrier. This non-blocking variant can be used to overlap the computation and communication among threads. It allows each thread to do its local computations while waiting for other threads to finish the previous section of a program. Hence it is suggested to use split-phase barrier wherever it is possible for a thread to do local computation in parallel to waiting for other threads to finish the previous section of program.

**Bulk Data Transfer Techniques**

UPC provides different string handling functions depending on the residence of source and destination buffers in either private or shared address space. These functions are *upc_memcpy*, *upc_memget*, *upc_memput* and *memcpy*. Programmers can employ techniques in a program such as a thread copies the bulk data from a remote memory location and then operate on it locally as compared to accessing the remote memory location immoderately. It is the most basic optimization technique, where a thread copies a remote data to its local buffer to avoid excessive remote memory accesses [5]. This can result in coarse-grained communication compared to fine-grained communication in program. It is especially very useful when a function takes shared variable as an input argument and it does not modify it. So by copying the value of this shared variable to a local variable, a thread can avoid accessing remote memory location very frequently [5].

**Load Balancing**

UPC provides constructs that allow each thread to compute on data it has affinity to. To achieve an equal work load distribution among threads, UPC provides the concept of shared arrays. The programmer can efficiently distribute the data among threads that avoids wasting the machine cycles by leaving a thread idle in a function or program. Programmers can then use the execution path control semantics for threads to direct each thread to work on data it is designated for. Hence an efficient load balancing can result in productive use of machine cycles.

We have now covered some very core concepts of UPC. We will now move on to next chapter where the explanation about UPC compilers is provided.

# 4 UPC Compilers

There are various commercial and open source compilers for programs written in UPC. They generally are comprised of a source to source translator and a run-time system. We have used three variants of UPC compilers. Each of these is discussed in subsequent sections of this chapter. These UPC compilers are:

1. Berkley UPC Compiler
2. GNU UPC Compiler
3. Cray UPC Compiler

## 4.1 Berkley UPC Compiler

Berkley UPC is an open source portable compiler for UPC programs unlike other commercially available UPC compilers which are system dependent. Few examples of these commercial UPC compilers are HP UPC compiler, SGI UPC compiler and Cray UPC compiler. It has been observed that the performance of a UPC program compiled with HPC compiler and executed on HPC Alpha server is comparable to that of a UPC [2]. Therefore it was required to develop an open source as well as a portable UPC compiler that matches the performance achieved using commercial UPC compilers.

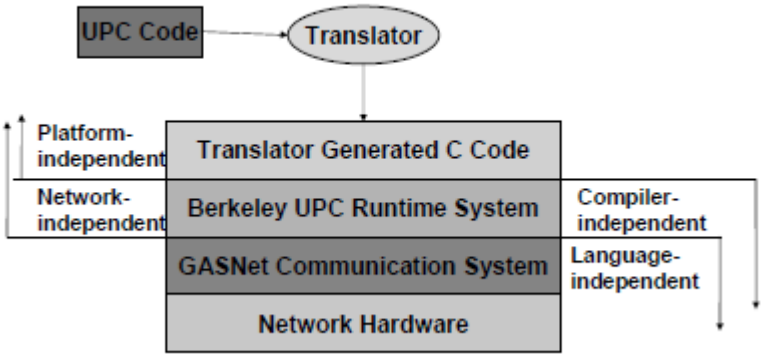


Figure 4.1(a): Berkley UPC Compiler architecture [2]

There are three components that make up a Berkley UPC compiler. These are source-to-source translator, run-time system and a communication system based on GASNet. GASNet is a network independent low level networking layer that is targeted for PGAS languages. The process of compilation starts by invoking the Berkley UPC compiler, which uses its source-to-source translator to convert a program written in UPC to a conventional ANSI C program augmented with calls to UPC run-time library routines [2]. Once the UPC code is translated to traditional ANSI C code, it is compiled by available C compiler on the target system and all the UPC run-time library routines are linked to UPC's run-time system. The UPC's run-time system is responsible for the allocation of data on shared space, calls to UPC routines, shared address resolution and executing program in parallel fashion [2]. All the communication for accessing shared space is handled by GASNet communication system.

A UPC program can be compiled by invoking Berkley UPC compilation command *upcc*. A regular .c file can also contain the UPC code as shown in compilation example below:

upcc –o main file1.upc file2.c

There are few important compilation options provided with upcc command. These are:

i. Specifying number of threads at compile time using option –T=*numThreads*
ii. Specifying default amount of shared memory in megabytes per UPC thread using option -

shared-heap=*sharedMemoryPerThread*

An example of compiling a program for 4 threads with each thread requiring an amount of shared memory equals to 256MB is shown below:

upcc -T=4 -shared-heap=256 -o main file1.upc file2.c

A program compiled with Berkley UPC compiler can be executed using *upcrun* command. A UPC program compiled with BUPC without static threads, can be executed using following command:

uprun –n 16 ./main

where –n option specified number of dynamic threads to be spawned.

## 4.2 GNU UPC Compiler

The GNU UPC system consists of a GCC UPC compiler and a run-time system. Like Berkley UPC, it is also an open source UPC compiler. GCC UPC inherits the functionality of conventional GCC compiler and provides additionally the compilation support for UPC programs. GNU UPC is available for various Linux distributions and has support of various Intel architectures including several Cray platforms. UPC programs can be compiled using GNU UPC compiler by invoking *gupc* command, as shown below:

gupc main.upc -o main

GUPC also provides support for compiling a UPC program for fixed number of threads by using -fupc-threads-N option, where N represents the number of threads. An example of compiling a program for 4 threads using GCC UPC is shown below:

gupc -fupc-threads-4 program.upc -o main

A program compiled using GNU UPC compiler can be executed similar to a serial C program compiled using gcc. However it provides option to specify number of threads at run-time using –n argument when a program is not compiled for static number of threads. A program compiled with GNU UPC compiler for dynamic number of threads is executed with 16 threads by following command:

./main –n 16

## 4.3 Cray UPC Compiler

Cray systems also provide support for UPC in its compiler. To compile a UPC program using Cray compiler, specific programming environment module is required to be loaded on Cray machine. A UPC program can be compiled using Cray's C compiler by specifying -h upc option, as shown below:

cc -h upc -o main main.c

Cray compiler also allows specifying number of threads at compiler time using -X option. An example of compiling a program for 4 threads using Cray UPC compiler is shown below:

cc -h upc -X 4 -o main main.c

A program compiled with Cray UPC compiler can be executed using *aprun* command available on Cray machines. However for executing a program compiled for static number of threads using Cray UPC compiler, same number of threads must be specified to the *aprun* command.

# 5 Porting CMD Code to Unified Parallel C

## 5.1 Introduction

Molecular Dynamics simulates interaction between molecules placed at different positions in a system. In our CMD code, the system of molecules is named phasespace. The phasespace is a bin which contains a large molecule container. The molecule container is divided into cells where each cell contains number of molecules. The relationship between the phasespace, molecule container and molecule cells is shown in figure 5.1(a):

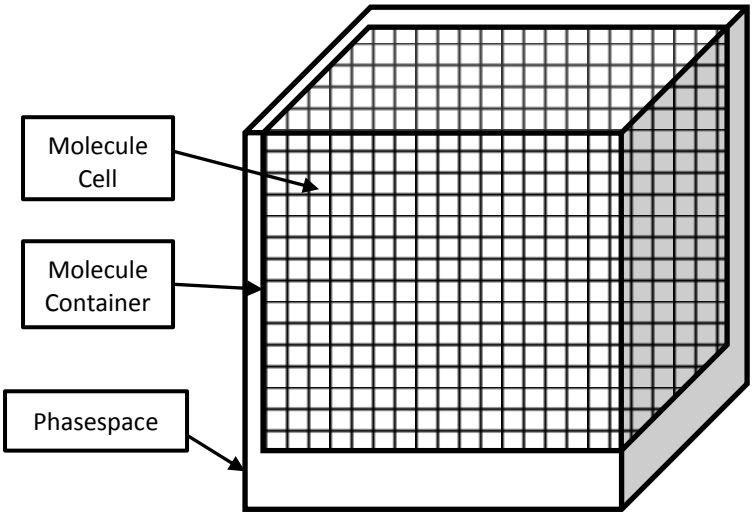


Figure 5.1(a): Hierarchy of Phasespace, Molecule Container and Molecule Cells

Our CMD simulation code is comprised of nine major steps, as shown in figure 5.1(b). These steps include:

1. Reading input parameters
2. Phase-Space Initialization
3. Grid Generator
4. Calculation of Initial Ensemble Values
5. Reset Forces and Momenta
6. Pre-Force Integration
7. Lennard-Jones Force and Potential Calculation
8. Post-Force Integration
9. Calculation of Ensemble Values

The pre simulation setup involves steps 1 through 4, while the main simulation loop incorporates steps 5 through 9. The pre simulation setup includes allocation of fix number of molecules to the molecule cells in the phasespace and then initializing the positions and velocities of these molecules. The main loop complies with the core simulation of our CMD code. It involves computation of forces acting on all molecules in the system and then Netwton's equations of motion are integrated to advance the simulation. The main loop is repeated until the time evolution of phasespace for the desired length of simulation time is being computed. All of these steps are discussed in detail in later sections of this chapter.

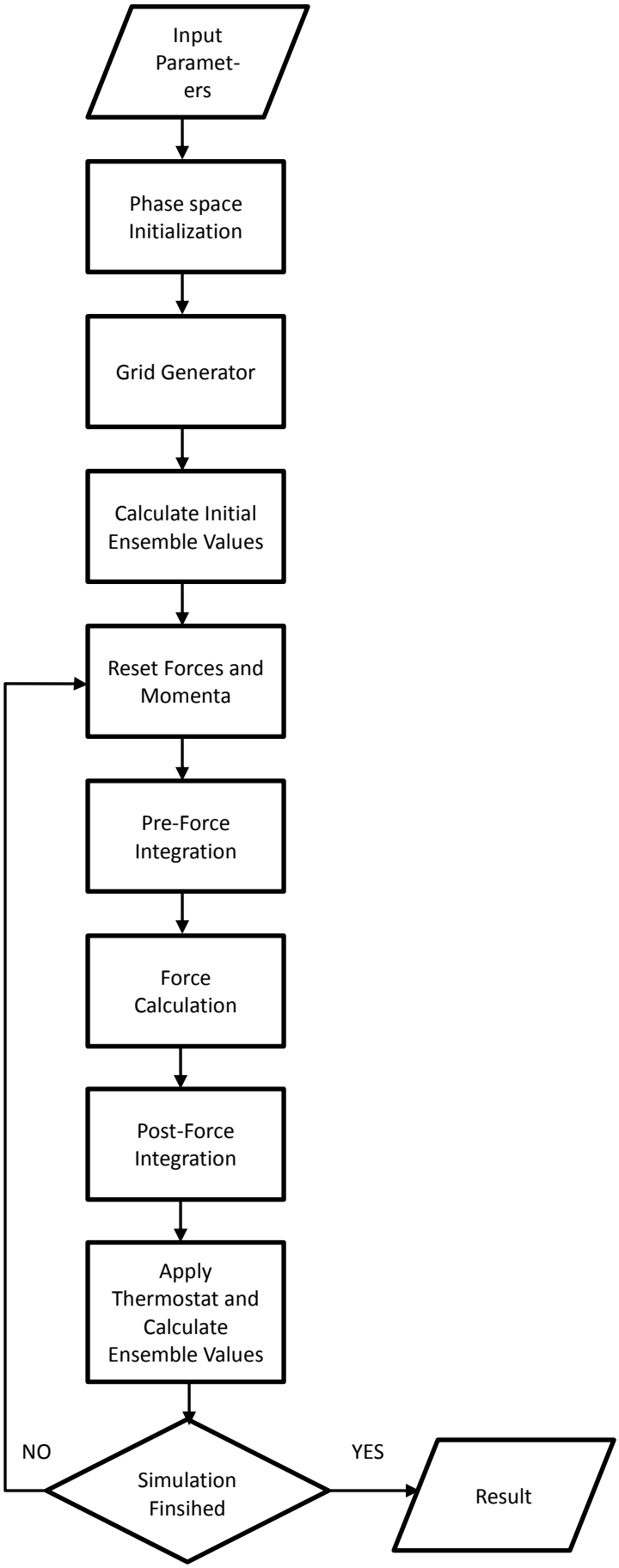


Figure 5.1(b): Flow of CMD Application

## 5.2 Reading Input Parameters

This step involves reading values which are mandatory to start the simulation. These values are provided to the program as run-time arguments and include cutoff radius, domain type, main loop control parameters, mass of a molecule and initial ensemble values (and few more). The purpose of these values is defined below:

*Cutoff Radius:* Cutoff radius is used to calculate the interactions between the molecules. If two molecules have a distance between them less than the cutoff radius, only then the interaction among these molecules is computed. The interaction among molecules is composed of Lennard Jones interaction force calculation.

*Domain type:* It is a string and can either have a value of Rectangle or Cube. According to the selected domain type, the initial positions and velocities of molecules are determined in the phasespace.

*Main loop control parameters:* These include simulation start and end times, and time-step length. They are used to control the duration or number of iterations of main loop along with time-step length.

*Mass of molecule:* It is a constant value and used to calculate the global momentum, global mass and acceleration of each molecule (which is then used to compute the velocities and positions of the molecules).

*Initial Ensemble values:* Ensemble values are core components of our simulation. They are required and computed throughout the simulation. These values include number of molecules in system, volume, energy, total kinetic energy and potential energy of interaction of molecules etc.

*Number of Threads:* This parameter is used by UPC's run-time system to create specific number of threads to execute the program in a SPMD fashion.

## 5.3 Phase Space Initialization

Phase space initialization involves allocation of memory dynamically on shared space for molecules and cells, distribution of cells among threads in a spatially coherent manner, and defining neighbors for each cell.

In our CMD code, phasespace, molecule container and molecule cells are declared as structures. The structure of phase space has only one element which is pointer to the structure of molecule container. The molecule container holds the molecules which are distributed among molecule cells. Both phase space and molecule container are allocated in private space in UPC's global address space. Hence each thread has private copy of these structures. The structure of molecule container beside other parameters contains private pointers to shared space, *cells* and *molecules*, which point to dynamically allocated memory on shared space for molecule cells and molecules respectively. The structure of molecule cell contains an array of private pointers to shared space for accessing neighbor cells by a thread. It also contains a local pointer and a private pointer to shared space, both of which point to the memory area belonging to the molecules of this cell. The purpose of both of these pointers to point to same memory area is when a thread which has affinity to a cell it can access its molecules using a local pointer whereas when a thread does not has affinity to a cell, it needs to access its molecules using pointer to shared space. The structure of cell also contains a lock which is required to be acquired by a thread when this thread needs to access the cell. The structures of molecule container and molecule cells are shown in figure 5.3(a) and 5.3(b) respectively.

**Molecule Container Structure**

```
typedef struct molecule_container_t {
   real cutoff_radius;
   long num_cells;
   long num_cells_per_dim[3];
   real cell_size[3];
   shared [BLOCK_QUALIFIER] molecule_cell_t *cells; /* Pointer to molecule cells to be
                                                       allocated dynamically on shared space */
   shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *molecules;  /* Pointer to molecules to be
                                                       allocated dynamically on shared space */

} molecule_container_t;
```

Figure 5.3(a): Structure of Molecule Container

**Molecule Cell Structure**

```
typedef struct molecule_cell_t molecule_cell_t;
struct molecule_cell_t {
        long id_neighbour[13];

        /* Private pointers to shared space to access neighboring cells */
         shared [BLOCK_QUALIFIER] molecule_cell_t *neighbours[13];

        /* Local Pointer to access molecules on local portion of shared space */
        real *data;

        /* Private Pointer to shared space to access remote molecules on shared space */
        shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *sdata;

        /* Lock of Cell */
        upc_lock_t * lock_mycell;
};
```

Figure 5.3(b): Structure of Molecule Cell

When the phasespace initialization routine (*psp_init*) is called from the main function, it follows the sequence of function calls as shown in figure 5.3(c). The routine of *psp_init* only calls the routine *mc_init* inside it and does nothing else. The function *mc_init* dynamically allocates the memory on shared space for molecules and cells, and set the private pointers to shared space (inside the structure of molecule container) to point to these allocated memory areas. The *mc_init* routine is shown in figure 5.3(d).


Figure 5.3(c): Sequence of calls for Phase Space initialization

The allocation of cells is done with the blocking factor of BLOCK_QUALIFIER. Hence each thread blocks the BLOCK_QUALIFIER number of cells on its local shared portion of the distributed shared space, as shown in figure 5.3(e). Similarly the allocation of molecules is done with the blocking factor of BLOCK_SIZE times BLOCK_QUALIFIER. Therefore each thread blocks the BLOCK_SIZE times BLOCK_QUALIFIER number of molecules on its local shared portion of the distributed shared space, as shown in figure 5.3(f). Here BLOCK_SIZE represents the maximum number of molecules that can be filled inside one molecule cell.

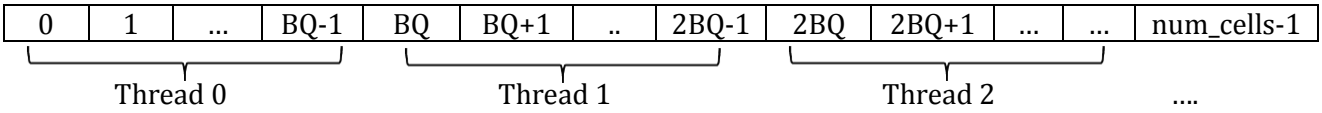


Figure 5.3(e): Allocation of molecule cells on shared space and their affinity to each thread

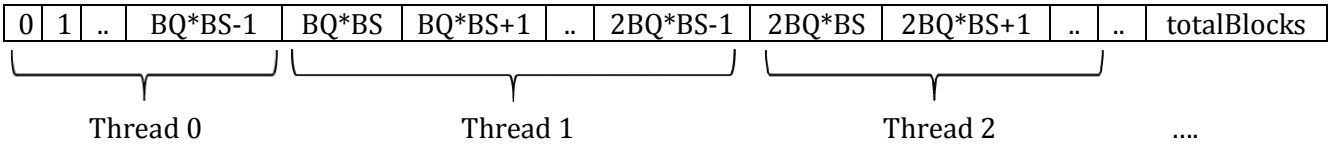


Figure 5.3(f): Allocation of molecules on shared space and their affinity to each thread

**Phase Space or Molecule container initialization**

```
void mc_init(ensemble_t *ensemble, molecule_container_t *mc) {
        int totalBlocks;
        int minBlocks;
        real box_length = pow(ensemble->V, 1./3.);
        mc->num_cells = 1;
        assert(mc->cutoff_radius > 0);
         int d;
         for(d = 0; d < 3; d++) {
                domain.L[d] = box_length;
                mc->num_cells_per_dim[d] = floor(domain.L[d] / mc->cutoff_radius) + 2; /* 2 more
                                cells in each direction to handle boundary conditions and halo */
                assert(mc->num_cells_per_dim[d] > 2);
                mc->num_cells *= mc->num_cells_per_dim[d];
                mc->cell_size[d] = domain.L[d] / (mc->num_cells_per_dim[d] - 2);
                assert(mc->cell_size[d] <= domain.L[d]);
        }
        assert(mc->num_cells>0);

  /* Considering the worst case of filling only one molecule in each molecule block except the last one,
   * THEN there is need of more number of molecule blocks to fill total molecules. Below equation
   * considers the worst case */

        minBlocks = mc->num_cells;
        totalBlocks = (mc->num_cells - 1) + ceil((ensemble->N + 1 - mc->num_cells)/CELL_CAPACITY);
        if(totalBlocks < minBlocks)
                totalBlocks = minBlocks;

  /* Allocating memory for molecule blocks and cells -- each cell contains ONLY 1 molecule block
   * Number of Blocks = number of Cells*/

         mc->molecules = (shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *)  upc_all_alloc(ceil(
                totalBlocks/BLOCK_QUALIFIER), BLOCK_SIZE * BLOCK_QUALIFIER * sizeof(real));
        mc->cells = (shared [BLOCK_QUALIFIER] molecule_cell_t *) upc_all_alloc(mc->num_cells /
                BLOCK_QUALIFIER, BLOCK_QUALIFIER * sizeof(molecule_cell_t));
        assert(mc->molecules != NULL);
        assert(mc->cells != NULL);



        long i, j, k;
```

```
            for(i = 0; i < mc->num_cells_per_dim[0]; i++) {
               for(j = 0; j < mc->num_cells_per_dim[1]; j++) {
                  for(k = 0; k < mc->num_cells_per_dim[2]; k++) {
                     long cell_id;
                      cell_id = get_cell_id( i, j, k ,mc);

                      /* If the CELL has affinity to current thread, then cast the shared pointer to local
                          pointer */
                     if(MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
                        molecule_cell_t * cell = (molecule_cell_t *) &mc->cells[cell_id];
                        mcell_alloc(cell, mc->molecules); /* Pass pointer to shared memory*/
                        cell->lock_mycell = upc_global_lock_alloc(); /*Initialize lock */

                     /* Setting neighbor cells for each cell not shown here*/
         }
        }
     }
   }
}
```

Figure 5.3(d): *mc_init* routine

After allocation of cells and molecules on shared space, the *mc_init* routine calls *mcell_alloc* routine and passes to it the pointer to cell that has affinity to the calling thread. The *mcell_alloc* routine is responsible for setting pointers of a cell, local and shared, such that they point to the same molecule block which lies in the local portion of the shared space of calling thread. A molecule block is the memory area for a single molecule cell to hold the fix number of molecules. Each thread can have many cells which lie in its local portion of shared space and these cells point to the molecule blocks that also reside in its local portion of the shared space. Hence each thread calls the *mcell_alloc* routine for the cells it has affinity to and initializes the pointers of those cells to point to the molecule blocks that also have affinity to this thread, as shown in figure 5.3(g). This is achieved by calling the *molecule_block_alloc* routine inside *mcell_alloc* routine, shown in figure 5.3(h).

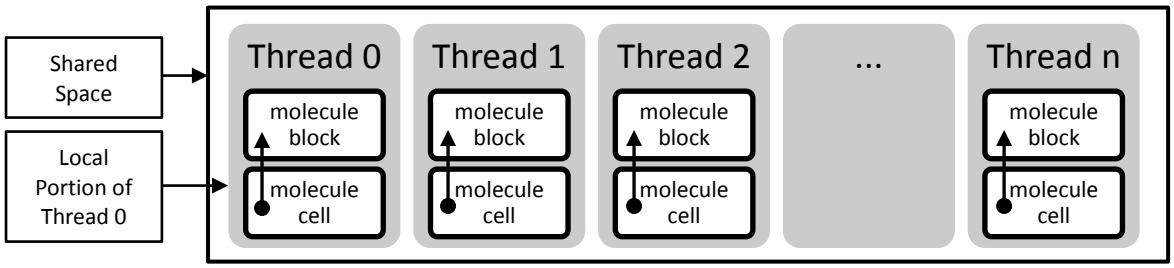


Figure 5.3(g): Each thread initializes pointers in cell structure to point to local portion of molecules (molecule block) allocated on shared space

The routine of *get_cell_id* called inside the routine *mc_init* is responsible for distribution of cells among threads in a spatially coherent manner. It generates a unique cell ID for blocking the spatially coherent cells in the local portion of shared space for each thread. The *get_cell_id* routine is shown in figure 5.3(j). The number of spatially coherent cells is determined using constants blockDimx, blockDimy and blockDimz. The *get_cell_id* routine utilizes these constants intelligently to generate consecutive IDs for spatially coherent cells and then these consecutive cell IDs are blocked by a thread. Given the 3D coordinates, it generates the successive integer IDs for spatially coherent cells, as shown in figure 5.3(i) for four threads. The function *get_cell_id* can be viewed as a mapping function which maps the cell coordinate values to a unique ID.

**Definition of Constants to be used in *molecule_block_alloc* routine (molecule_block.h)**

```
#define blockDimx 3
#define blockDimy 3
#define blockDimz 3
#define BLOCK_SIZE (CELL_CAPACITY * 14 + 1) /* Size of a single molecule block */
#define BLOCK_QUALIFIER (blockDimx*blockDimy*blockDimz)  /* Consecutive molecule cells per
                                                                thread*/
#define START_OFFSET (MYTHREAD * BLOCK_SIZE * BLOCK_QUALIFIER) /*Start offset for each
                                        thread for the allocation of molecules to its cells*/
```

***molecule_block_alloc* Routine (molecule_block.c)**

```
int block_offset = 0; /* Offset for current molecule block within block qualifier */
int block_position = 0; /* Position of molecule block relative to starting address of shared space which
                                                                has affinity to MYTHREAD */
int allocated_blocks = 0;
int offset = 0;

/* Called ONLY by a thread which has affinity to this block or cell */
void molecule_block_alloc(real **block, shared [BLOCK_SIZE * BLOCK_QUALIFIER] real **sblock,
shared [BLOCK_SIZE * BLOCK_QUALIFIER] real * pointer) {
        /* Identify molecule block position */
        if(block_offset == BLOCK_QUALIFIER){
                offset++;
                block_offset=0;
        }
        block_position = block_offset * BLOCK_SIZE + offset * (THREADS * BLOCK_SIZE *
                                BLOCK_QUALIFIER); /* Position within consecutive blocks per thread */
        block_offset++;

        /* Point to the next molecule block current thread has affinity to */
        shared [BLOCK_SIZE * BLOCK_QUALIFIER] real * next_block = &pointer[START_OFFSET +
                                                                block_position];

        /* If next block has affinity to current thread, then cast shared to private Pointer */
        if (MYTHREAD == upc_threadof(next_block)) {
                (*sblock) = next_block; // pointer to shared
                (*block) = (real *) next_block; //cast shared to private

                /* Set flags to zero of newly allocated block */
                memset((*block), 0, CELL_CAPACITY * sizeof(real));
                allocated_blocks++; // count total allocated blocks
        }
}
```

Figure 5.3(h): Molecule block allocation

The *mc_init* routine is also responsible for allocation of lock for each cell dynamically on shared space and setting the neighbors of each cell. It controls the execution path for each thread, such that every thread works on the data it has affinity to. The selection of different execution path for each thread is achieved using UPC's built in function *upc_threadof* inside the if statement. Each thread calls the *upc_global_lock_alloc* routine for the cells allocated on its local portion of shared space to allocate a separate lock for each of its cells. Similarly each thread sets the neighbors of the cells it has affinity to.

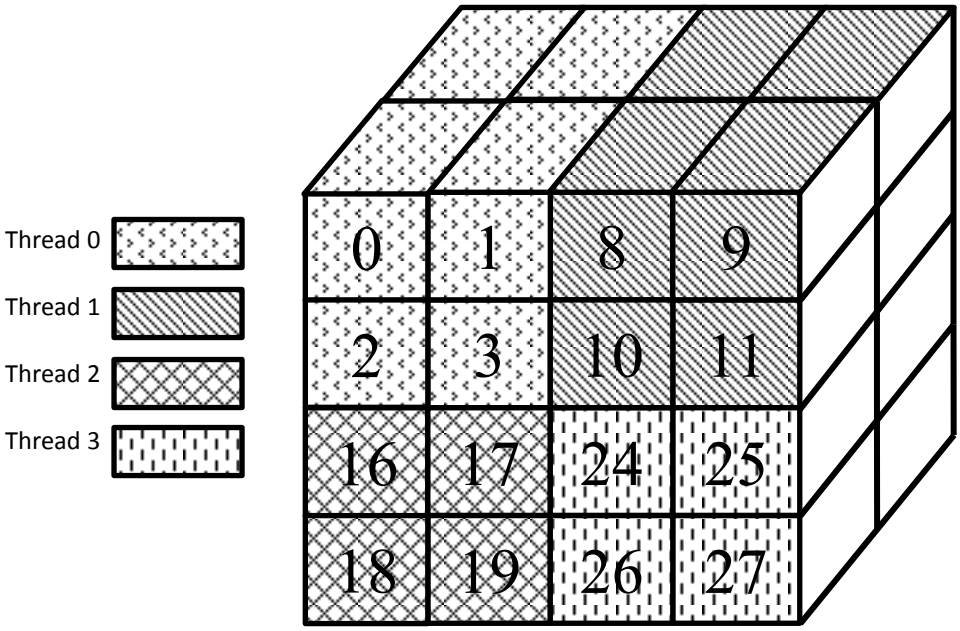


Figure 5.3(i): Consecutive integer cell ID generation for spatially coherent cells by routine *get_cell_id*

**Routine for consecutive IDs generation for spatially coherent cells**

```
static long get_block_number(long blockIdx, long blockIdY, long blockIdZ, molecule_container_t *mc) {
        long block_number = mc->num_cells_per_dim[1] * mc>num_cells_per_dim[2] /
                                (blockDimy*blockDimz) * blockIdx  + mc->num_cells_per_dim[2] /
                                blockDimz * blockIdY + blockIdZ;
        return block_number;
}

static void get_block_parameters(long x, long y, long z, long block_id[3], long block_indices[3]) {
        block_id[0] = x / blockDimx;
        block_id[1] = y / blockDimy;
        block_id[2] = z / blockDimz;
        block_indices[0] = x % blockDimx;
        block_indices[1] = y % blockDimy;
        block_indices[2] = z % blockDimz;
}

static long get_cell_id(long ix, long iy, long iz, molecule_container_t *mc) {
        long block_id[3], block_indices[3];
        long block_number;
        long index;

        if (ix < 0)
                ix += mc->num_cells_per_dim[0];
        else if (ix >= mc->num_cells_per_dim[0])
                ix -= mc->num_cells_per_dim[0];
        if (iy < 0)
                iy += mc->num_cells_per_dim[1];
        else if (iy >= mc->num_cells_per_dim[1])
                iy -= mc->num_cells_per_dim[1];
        if (iz < 0)
                iz += mc->num_cells_per_dim[2];
        else if (iz >= mc->num_cells_per_dim[2])
```

```
                iz -= mc->num_cells_per_dim[2];

        assert(mc->num_cells_per_dim[0]%blockDimx == 0 && mc->num_cells_per_dim[1] %
                blockDimy == 0 && mc->num_cells_per_dim[2]%blockDimz == 0);

        get_block_parameters(ix, iy, iz, block_id, block_indices);

        /* For 2D Block:
         * 1. block_size is blockDimy*blockDimz */
//      block_number = get_block_number(ix, block_id[1], block_id[2], mc);
//      index = block_number*blockSize+ blockDimz*block_indices[1]+block_indices[2];

        /*For 3D block:
         * 1. ix is now changed to block_id[0]=ix/blockDimx to get correct block number
         * 2. block_size is blockDimx*blockDimy*blockDimz
         * 3. Index calculation is summation of number of cells covered in:
         *      block_number*blockSize = blocks before current block
         *      blockDimy*blockDimz*block_indices[0] = current block but on DIFFERENT layer(s)
         *      blockDimz*block_indices[1] + block_indices[2] = current block on SAME layer
         *      */

        block_number = get_block_number(block_id[0], block_id[1], block_id[2], mc);
        index = block_number*blockSize + blockDimy*blockDimz*block_indices[0] +
                blockDimz*block_indices[1] + block_indices[2];

        return index;
}
```

Figure 5.3(j): *get_cell_id* routine

## 5.4 Grid Generation

Grid generation involves initialization of positions and velocities of the molecules and adding them to the phasespace. The process of grid generation is initiated by call to the function *grid_generator* inside main. The call to *grid_generator* routine follows a sequence of calls for the addition of molecules to cells. This sequence of calls is shown in figure 5.4(a). The *grid_generator* routine calls the routine *psp_add_molecule* and passes to it pointers to the molecule container and the molecule (with initialized position and velocity) to be added. The *psp_add_molecule* routine simply calls the *mc_add_molecule* routine with the same arguments. Inside *mc_add_molecule* routine, the different flow of execution for each thread is controlled in such a way that each thread is responsible for adding the molecules to the cells it has affinity to. Each thread cast the pointer-to-shared to a local pointer to access these cells, and hence calls the routine *mcell_add_molecule_local* passing to it local pointer to the cell located in its local portion of shared space. This in turn avoids access overhead due to use of pointer to shared space [1][2] which can be avoided by using local pointer in this case. The routine of *mc_add_molecule* is shown in figure 5.4(b).


Figure 5.4(a): Sequence of calls for addition of molecules to cells

```
void mc_add_molecule(molecule_t *m, molecule_container_t *mc){
   long cell_id = get_cell_index_from_coordinate(m->r[0], m->r[1], m->r[2], mc);
   if(MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
          molecule_cell_t *cell = (molecule_cell_t *)&mc->cells[cell_id];
          mcell_add_molecule_local(m, cell);
   }
}
```

Figure 5.4(b): *mc_add_molecule* routine

We have developed locality aware algorithms for most of the functions in our CMD code. These functions will be explained in the following sections.

## 5.5 Reset Forces and Momenta

This is the first step of main loop in CMD simulation, shown in figure 5.5 (a). The cycle of routines called in main loop is shown in figure 5.5(b). The process of resetting forces is initiated by call to routine *psp_reset_forces* inside main loop. Once the routine *psp_reset_forces* is called, it follows a sequence of calls shown in Figure 5.5(c). The routine of *mc_reset_forces_and_momenta* implements a locality-aware algorithm for resetting the forces of molecules in phasespace, as shown in figure 5.5(d). This is done by directing each thread to a different execution path such that each thread only reset forces of molecules inside the cells it has affinity to. Remember that the phasespace contains a molecule container which is divided into cells of molecules and these cells are distributed among threads in the phase space initialization step. For resetting the forces, each thread access its cells using local pointers to eliminate the access overhead introduced by pointers to shared space, as shown in figure 5.5(e). Please note that we introduced both local and shared pointers in the structure of molecule cell to point to same molecule block. This was done so that when a thread which has affinity to a cell, it can access its molecules using local pointer, whereas a thread which does not has affinity to a cell must access the molecules of this cell using pointer to shared space.

```
start_timer = timer();
// Start of main loop
for(stime = simulation.start_time; stime < simulation.end_time; stime += simulation.dt) {
      upcfprintf(stdout, "Current simulation time: %"PRIreal"\n", stime );
      psp_reset_forces_and_momenta(psp);

      upcfprintf(stdout, "Starting pre force integration step.\n" );
      psp_integrate_pref(psp, &simulation);

      /********************************* SYNC_POINT1 *********************************/
      upc_notify SYNC_POINT1;

      upcfprintf(stdout, "Starting force calculation.\n" );
      psp_calc_forces(psp, &ensemble);

      upcfprintf(stdout, "Starting post force integration step.\n" );
      psp_integrate_postf(psp, &simulation, &domain, &ensemble);

      psp_calc_ensemble_values(psp, &ensemble);
      psp_apply_thermostat(psp, &ensemble);
      psp_calc_ensemble_values(psp, &ensemble);
      ensemble_print_info(&ensemble);

      if( ENABLED == config.ascii_output) {
        upc_barrier SYNC_POINT7;
```

```
        if(MYTHREAD == 0 ) {
           FILE *fh;
           char filename[256];
           snprintf(filename,255, "psp-%"PRIreal".dat", stime);
           fh = fopen(filename, "w+");
           mc_print_ascii(psp->mc, fh);
           fclose(fh);
        }
        upc_barrier SYNC_POINT8;
     }
     timesteps++;
         } // end of main loop

end_timer = timer();
```

Figure 5.5(a): Main Simulation Loop

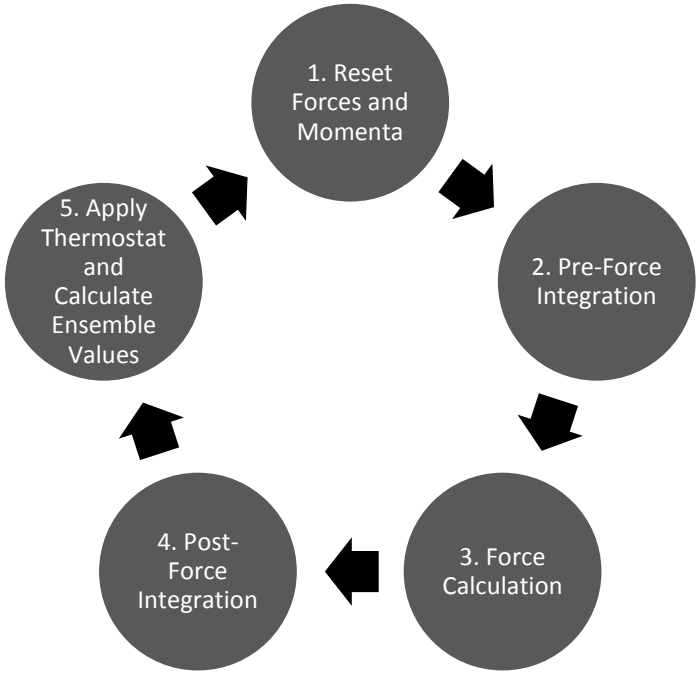


Figure 5.5(b): Cycle of routines called inside main simulation loop


Figure 5.5(c): Sequence of calls for resetting forces of molecules inside phase space

```
void mc_reset_forces_and_momenta(molecule_container_t *mc) {
  long i, j, k;
  for(i = 1; i < mc->num_cells_per_dim[0] - 1; i++) {
    for(j = 1; j < mc->num_cells_per_dim[1] - 1; j++) {
      for(k = 1; k < mc->num_cells_per_dim[2] - 1; k++) {
        long cell_id;
        cell_id = get_cell_id( i, j, k ,mc);
        if(MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
          molecule_cell_t * cell = (molecule_cell_t *) &mc->cells[cell_id];
```

```
            mcell_reset_forces_and_momenta(cell);
          }
        }
      }
    }
}
```

Figure 5.5(d): *mc_reset_forces_and_momenta* routine

```
/* Called ONLY by a thread which has affinity to this block or cell */
void molecule_block_reset_forces_and_momenta(real *block) {
    real *F = &block[4*CELL_CAPACITY];
   memset(F, 0, CELL_CAPACITY * 3 * sizeof(real));
}
```

Figure 5.5(e): *molecule_block_reset_forces_and_momenta* routine

## 5.6 Pre-Force Integration

Pre-force integration is the second step in main simulation loop. The process of pre-force integration is initiated by call to routine *psp_integrate_pref* inside main loop. Once the routine *psp_integrate_pref* is called, it follows a sequence of calls shown in Figure 5.6(a). The routine of *mc_integrate_pref* also implements a locality-aware algorithm for picking out different execution path for each thread, as shown in figure 5.6(b). Finally the routine *molecule_block_integrate_pref* evaluates the velocities and positions of molecules in phasespace. The routine *molecule_block_integrate_pref* is aided with cache optimizations by providing information to the compiler about pointer aliasing using the restrict keyword in ANSI C, shown in figure 5.6(c). This helps is not loading the value of 'F' twice in cache, by instructing the compiler that pointers 'F' and 'V' do not point to the same memory location.


Figure 5.6(a): Sequence of calls for pre-force integration

```
void mc_integrate_pref(molecule_container_t *mc, simulation_t *simulation) {
   long i, j, k;
   for(i = 1; i < mc->num_cells_per_dim[0] - 1; i++) {
      for(j = 1; j < mc->num_cells_per_dim[1] - 1; j++) {
         for(k = 1; k < mc->num_cells_per_dim[2] - 1; k++) {
            long cell_id;
            cell_id = get_cell_id( i, j, k ,mc);
            if(MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
               molecule_cell_t * cell = (molecule_cell_t *) &mc->cells[cell_id];
               mcell_integrate_pref(simulation, cell);
            }
         }
      }
   }
}
```

Figure 5.6(b): *mc_integrate_pref* routine

```
/* Called ONLY by a thread which has affinity to this block or cell */
void molecule_block_integrate_pref(simulation_t *simulation, real *block) {
   int d, i;
   real *restrict flags = block;
   real *restrict r = flags + CELL_CAPACITY;
```

```
    real *restrict F = r + 3 * CELL_CAPACITY;
    real *restrict v = F + 3 * CELL_CAPACITY;
    real a = 0.5 * simulation->dt / config.m;
            for(d = 0; d < 3; d++) {
                    for(i = 0; i < CELL_CAPACITY; i++) {
                        v[3*i+d] += flags[i] * a * F[3*i+d];
                        r[3*i+d] += simulation->dt * (v[3*i+d] + 0.5/config.m * F[3*i+d] * simulation->dt);
                    }
            }
}
```

Figure 5.6(c): *molecule_block_integrate_pref* routine

## 5.7 Lennard-Jones Force Calculation and Potential Integration

This is the third and most compute intensive step of simulation. In this step the forces acting upon each molecule in the phasespace is being computed. If any two molecules in the phasespace have distance among them less than the cut-off radius, provided as input parameter to simulation, then interaction among these molecules is computed. Each thread calculates the interaction of molecules residing in cells it has affinity to and integrates the lennard-jones potential for each interaction. The coordination among threads to calculate the global value of potential energy is done using reduction function available in collective library of UPC. There are three distributed shared arrays allocated statically with the blocking factor of 1 and having total number of elements equal to total number of threads, such that each thread has one element of this shared array which is allocated in its local portion of shared space. The purpose of these shared arrays is to store the number of computed interactions, potential energy and energy integrated by each thread for the molecules residing in cells each thread has affinity to. These shared arrays are named compute_interact, ensembleUPot and ensembleE respectively.

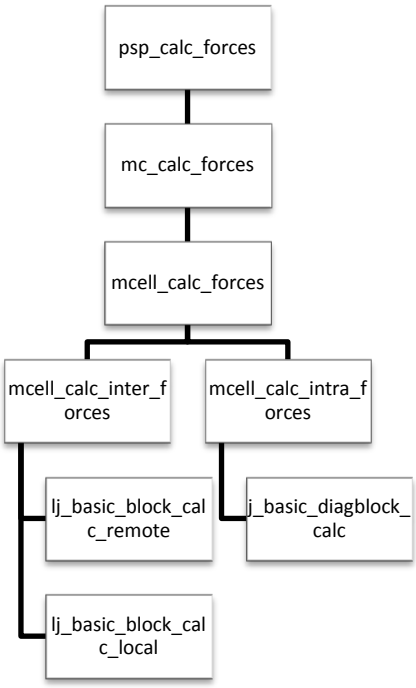


Figure 5.7(a): Hierarchy of calls for computation of molecules interactions

The process of Lennard-Jones Force calculation and potential integration is initiated with the call to *psp_calc_forces* routine from the main loop. It follows a hierarchy of calls shown in figure 5.7(a). In pre-force integration process, all threads calculate values of positions and velocities for molecules residing in cells located in their local portion of shared space respectively. The values of positions of molecules

```
shared [1] long compute_interact[THREADS];
shared [1] real ensembleUPot[THREADS];
shared [1] real ensembleE[THREADS];
long shared total_interact = 0;
shared real E_final;
shared real UPot_final;

void mc_calc_forces(molecule_container_t *mc, ensemble_t *ensemble) {
   long i;
   ensemble->E = 0.0;
   ensemble->U_pot = 0.0;
   ensemble->E_kin = 0.0;
   config.count=0;
   compute_interact[MYTHREAD] = 0;
   ensembleE[MYTHREAD] = 0;
   ensembleUPot[MYTHREAD] = 0;
   upc_wait SYNC_POINT1; /* Wait for matching notify operation to be executed by all threads */
   mc_update_halo(mc);
   for(i = 0; i < mc->num_cells; i++) {
      if(MYTHREAD == upc_threadof(&mc->cells[i])) {
           molecule_cell_t *cell = (molecule_cell_t *) &mc->cells[i];
           long * count_local = (long *) &compute_interact[MYTHREAD];
           real * U_Pot_local = (real *) &ensembleUPot[MYTHREAD];
           real * E_local =  (real *) &ensembleE[MYTHREAD];
           mcell_calc_forces(cell, count_local, U_Pot_local, E_local);
      }
           }
           /********************************* SYNC_POINT3 *********************************/
           upc_all_reduceD(&UPot_final, ensembleUPot, UPC_ADD, THREADS, 1, NULL, UPC_IN_MYSYNC |
                                   UPC_OUT_ALLSYNC);
           upc_all_reduceD(&E_final, ensembleE, UPC_ADD, THREADS, 1, NULL, UPC_IN_MYSYNC |
                                   UPC_OUT_ALLSYNC);
           upc_all_reduceL(&total_interact, compute_interact, UPC_ADD, THREADS, 1, NULL,
                                   UPC_IN_MYSYNC | UPC_OUT_ALLSYNC);
           if (MYTHREAD == 0)
              printf("Global Sum of Number of computed interactions: %lu\n", total_interact);
           ensemble->E = E_final;
           ensemble->U_pot = UPot_final;
}
```

Figure 5.7(b): *mc_calc_forces* routine

of neighbor cells are used in the force calculation process. Hence before starting process of force calculation, it must be assured that all threads have completed the pre-force integration process. This is asserted by the use of split-phase barrier call in the main simulation loop. Each thread calls the function *upc_notify* once it has completed the pre-force integration process, and before calling the update halo routine inside *mc_calc_forces* routine (shown in figure 5.7(b)) each thread calls the *upc_wait* function to wait for other threads to finish the pre-force integration process. The *mc_update_halo* routine further calls the routines *mc_update_corners*, *mc_update_edges* and *mc_update_surfaces*. These function copies the molecule cells and modifies the positions of their molecules. The update halo operation do not encounter a scenario where a cell which is used as destination in one copy operation can later also be used as source cell in another copy operation or vice versa. In such scenario, it must be assured that that one of these operations is completed before starting the other and also needs to take care of the sequence of these operations if it matters for final result. The CMD code has been tested for such scenarios and hence redundant synchronization is

avoided. Nevertheless, before actual force calculation is started it must be assured that process of update halo has been finished. This is achieved by providing a barrier synchronization using call to *upc_barrier* routine at the end of *mc_update_halo* routine, as shown in figure 5.7(c).

```
void mc_update_halo(molecule_container_t *mc) {
   /* copy cells to boundary to fullfill the periodic boundary condition. */
   /* we have 26 directions */
   /* 8 corners */
   mc_update_corners(mc);
   /* 12 edges */
   mc_update_edges(mc);
   /* 6 lateral surfaces */
   mc_update_surfaces(mc);

   /********************************** SYNC_POINT2 **********************************/
   upc_barrier SYNC_POINT2;
}
```

Figure 5.7(c): *mc_update_halo* routine

Inside *mc_calc_forces* routine, a locality-aware algorithm is implemented to assign each thread the calculation of forces acting on the molecules situated inside cells it has affinity to. We have utilized Newton's 3rd law of motion to avoid calculation of a force value twice. Newton's 3rd law of motion states that when a force is acted on a particle A by particle B, particle A acts an equal and opposite force on particle B. Although the use of Newton's 3rd law reduces the computation effort, it raises the synchronization requirement when accessing a molecule in the neighbor cell of a cell. This synchronization is provided by introducing a lock per cell instead of earlier used critical region synchronization by using a single atomic lock, as shown in the structure of molecule cell in Figure 5.3(b). Hence when the routines *mcell_calc_intra_forces* and *mcell_calc_inter_forces* are called inside *mcell_calc_forces* routine, pointers to lock s corresponding to the cells are passed as arguments. The routine *mcell_calc_forces* is shown in figure 5.7(e). It calls the *mcell_calc_intra_forces* and *mcell_calc_intra_forces* routines for computation of forces acting on molecules of its cell due to molecules in the same cell and due to molecules in the neighbor cell respectively. The *mcell_calc_intra_forces* routine calls the routine *lj_basic_diagblock_calc* and passes to it local pointer to the molecule block of this cell instead of shared pointer to overcome access overhead due to use of shared pointer. The *mcell_calc_inter_forces* routine is called for all surrounding neighbor cells of a cell, as shown in figure 5.7(d). It further calls either the routine *lj_basic_block_calc_local* or *lj_basic_block_calc_remote* depending on the locality of the neighbor cells. If the neighbor cell has also affinity to the calling thread, it calls the routine *lj_basic_block_calc_local*, while if neighbor cell does not have affinity to the calling thread, it calls the routine *lj_basic_block_calc_remote*. Inside routine *lj_basic_block_calc_local* both cells are accessed using local pointers whereas in *lj_basic_block_calc_remote* routine, neighbor cell is accessed using pointer to shared space.

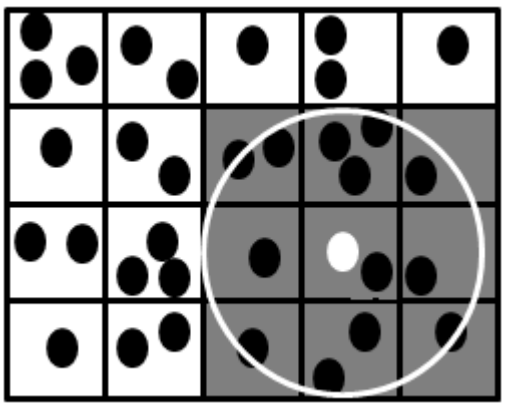

Figure 5.7(d): Interaction among molecules of a cell and its neighbor cells lying within cutoff radius

```
* Called ONLY by a thread which has affinity to this cell */
void mcell_calc_forces(molecule_cell_t *cell, long * count_local, real * U_Pot_local, real * E_local) {
   mcell_calc_intra_forces(cell, count_local, U_Pot_local, E_local);
   shared [BLOCK_QUALIFIER] molecule_cell_t *neighbour_cell;
   int i;
   for (i = 0; i < 13; i++) {
          neighbour_cell = cell->neighbours[i];
          assert(neighbour_cell != NULL);
          mcell_calc_inter_forces(cell, neighbour_cell, count_local, U_Pot_local, E_local);
   }
}

/* Called ONLY by a thread which has affinity to this cell */
void mcell_calc_intra_forces(molecule_cell_t *cell, long * count_local, real * U_Pot_local, real * E_local) {
  real *block1;
  block1 = cell->data;
  lj_basic_diagblock_calc(block1, count_local, U_Pot_local, E_local, cell->lock_mycell);
}

/* Called ONLY by a thread which has affinity to the CELL 1 */
void mcell_calc_inter_forces(molecule_cell_t *cell1, shared [BLOCK_QUALIFIER] molecule_cell_t *cell2,
long * count_local, real * U_Pot_local, real * E_local) {

  real *block1;
  block1 = cell1->data;

  if(MYTHREAD == upc_threadof(cell2)) {
          real *block2;
          block2 = cell2->data;
          lj_basic_block_calc_local(block1, block2, count_local, U_Pot_local, E_local, cell1->lock_mycell,
                                                                                  cell2->lock_mycell);
  }
  else {
          shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *block2;
          block2 = cell2->sdata;
          lj_basic_block_calc_remote(block1, block2, count_local, U_Pot_local, E_local, cell1->lock_mycell,
                                                                                  cell2->lock_mycell);
          }
}
```

Figure 5.7(e): *mcell_calc_forces*, *mcell_calc_intra_forces* and *mcell_calc_inter_forces* routines

Routines *lj_basic_diagblock_calc*, *lj_basic_block_calc_local* and *lj_basic_block_calc_remote* computes interaction among molecules only when the distance between two molecules is less than the cutoff radius. This is shown as a circle with center at the position of one molecule and having radius equals to cutoff radius in figure 5.7(d). As it can be seen in this figure that only three molecules lie in the range of cutoff radius of white molecule, the interaction of this molecule is only computed with these three molecules [9]. In routine *lj_basic_diagblock_calc*, a thread needs to acquire lock of the cell before modifying forces of its molecules. It simply calls the routine *upc_lock* passing pointer to the lock of this cell. In routines *lj_basic_block_calc_local* and *lj_basic_block_calc_remote*, a thread needs to acquire locks on both cells it has to calculate interaction among molecules of. It is important to highlight here a scenario when there is a cycle among threads waiting to acquire a lock. Consider a case when two threads need to access same two cells, one thread has acquired a lock on one cell and other thread has acquired a lock on other cell. Both of these threads will be blocked indefinitely if they try to acquire lock using blocking call *upc_lock* as they both will be waiting for other cell to release the lock. To avoid

such situations, we have utilized the non-blocking version of acquiring locks provided by upc, which is *upc_lock_attempt*. The routine *acquire_locks* acquires locks on both of the cells it needs to modify, if lock only on one of the cell is acquired it releases the acquired lock. Hence it prevents deadlock among threads.

Once all threads have computed interactions of molecules of their respective local cells and stored the results in the local elements of shared arrays, a collective call is made to UPC's reduction function *upc_all_reduceD*. With this reduction function, each thread coordinates to the other threads the results of its computation. A global sum of results of individual threads is stored by the reduction call in the shared variables (total_interact, E_final and UPot_final) to give the total computed interactions and integrated potential energies. The synchronization for reading or writing the shared space for reduction call is achieved by using appropriate synchronization flags as parameter to the reduction function.

```
void lj_basic_block_calc_local(real *block1, real *block2, long *restrict count_local, real *restrict
U_Pot_local, real *restrict E_local, upc_lock_t * lock_cell1, upc_lock_t * lock_cell2) {
  long i;
  real U_pot = 0.;

  /* Pointers to molecule block residing in local portion of shared space */
  real *restrict flags1 = block1;
  real *restrict r1 = flags1 + CELL_CAPACITY;
  real *restrict F1 = r1+ 3 * CELL_CAPACITY;
  real *restrict v1 = F1 + 3 * CELL_CAPACITY;

 /* Pointers to neighbor molecule block residing in local portion of shared space */
  real *restrict flags2 = block2;
  real *restrict r2 = flags2 + CELL_CAPACITY;
  real *restrict F2 = r2 + 3 * CELL_CAPACITY;
  real *restrict v2 = F2 + 3 * CELL_CAPACITY;

  for(i = 0; i < CELL_CAPACITY; i++) {
        if( flags1[i] == 0.)
        continue;
        long j;
        for(j = 0; j < CELL_CAPACITY; j++) {
          if( flags2[j] == 0.)
           continue;

          real dr[3];
          real dr2 = 0.;
          int d;
          for( d = 0; d < 3; d++ ) {
                dr[d] = r2[3*j+d] - r1[3*i+d];
                dr2 += dr[d] * dr[d];
          }
          if( dr2 > config.cutoff_radius_sq )
                continue;
          real invdr2 = 1. / dr2;
         config.count++;
         (*count_local)++;

          /* Lennard Jones interaction forces */
          real lj6 = sigma2 * invdr2;
          lj6 = lj6 * lj6 * lj6;
```

```
            real lj12 = lj6 * lj6;
            real lj12m6 = lj12 - lj6;
            real u_pot  = epsilon24 * lj12m6;
            real factor = epsilon24 * (lj12 + lj12m6) * invdr2;
#ifndef NDEBUG
        if(isnan(factor)) {
          upcfprintf(stderr, "Blocks\n");
          upcfprintf(stderr, "i: %lu, j: %lu, dr2: %"PRIreal", invdr2: %"PRIreal"\n", i, j, dr2, invdr2);
          molecule_block_print_ascii(block1, stderr);
          molecule_block_print_ascii(block2, stderr);
        }
#endif

            /* Start of Critical Section*/
            acquire_locks(lock_cell1, lock_cell2);
            for( d = 0; d < 3; d++ ) {
            F1[3*i+d] -= factor * dr[d];
            F2[3*j+d] -= F1[3*i+d];
            }
            upc_unlock(lock_cell1);
            upc_unlock(lock_cell2);
            /* End of Critical Section*/
            U_pot += u_pot;
          }
   }
   *E_local += U_pot;
   *U_Pot_local += U_pot;
}

void lj_basic_block_calc_remote(real *block1, shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *block2,
long *restrict count_local, real *restrict U_Pot_local, real *restrict E_local, upc_lock_t * lock_cell1,
upc_lock_t * lock_cell2) {
   long i;
   real U_pot = 0.;

   /* Local pointers to access molecule block of cell which has affinity to this thread */
   real *restrict flags1 = block1;
   real *restrict r1 = flags1 + CELL_CAPACITY;
   real *restrict F1 = r1+ 3 * CELL_CAPACITY;
   real *restrict v1 = F1 + 3 * CELL_CAPACITY;

   /* Pointers to shared space to access molecule block of neighbor cell */
   shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *restrict flags2 = block2;
   shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *restrict r2 = flags2 + CELL_CAPACITY;
   shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *restrict F2 = r2 + 3 * CELL_CAPACITY;
   shared [BLOCK_SIZE * BLOCK_QUALIFIER] real *restrict v2 = F2 + 3 * CELL_CAPACITY;

   for(i = 0; i < CELL_CAPACITY; i++) {
         if( flags1[i] == 0.)
           continue;
         long j;
         for(j = 0; j < CELL_CAPACITY; j++) {
           if( flags2[j] == 0.)
                 continue;
```

```
            real dr[3];
            real dr2 = 0.;
            int d;
            for( d = 0; d < 3; d++ ) {
                    dr[d] = r2[3*j+d] - r1[3*i+d];
                    dr2 += dr[d] * dr[d];
            }
            if( dr2 > config.cutoff_radius_sq )
                    continue;
            real invdr2 = 1. / dr2;
            config.count++;
             (*count_local)++;


            /* Lennard Jones interaction forces */
            real lj6 = sigma2 * invdr2;
            lj6 = lj6 * lj6 * lj6;
            real lj12 = lj6 * lj6;
            real lj12m6 = lj12 - lj6;
            real u_pot  = epsilon24 * lj12m6;
            real factor = epsilon24 * (lj12 + lj12m6) * invdr2;
#ifndef NDEBUG
        if(isnan(factor)) {
           upcfprintf(stderr, "Blocks\n");
           upcfprintf(stderr, "i: %lu, j: %lu, dr2: %"PRIreal", invdr2: %"PRIreal"\n", i, j, dr2, invdr2);
           molecule_block_print_ascii(block1, stderr);
           molecule_block_print_ascii_shared(block2, stderr);
        }
#endif
           /* Start of Critical Section*/
           acquire_locks(lock_cell1, lock_cell2);
           for( d = 0; d < 3; d++ ) {
              F1[3*i+d] -= factor * dr[d];
              F2[3*j+d] -= F1[3*i+d];
           }
           upc_unlock(lock_cell1);
           upc_unlock(lock_cell2);
           /* End of Critical Section*/
           U_pot += u_pot;
     }
   }
   *E_local += U_pot;
   *U_Pot_local += U_pot;
}

void lj_basic_diagblock_calc(real *block, long *restrict count_local, real *restrict U_Pot_local, real
*restrict  E_local, upc_lock_t * lock_cell) {
   long i;
   real U_pot = 0.;

   /* Local pointers to access molecule block of cell which has affinity to this thread */
   real *restrict  flags = block;
   real *restrict r = flags + CELL_CAPACITY;
   real *restrict F = r + 3 * CELL_CAPACITY;
   real *restrict v = F + 3 * CELL_CAPACITY;
```

```
   for(i = 0; i < CELL_CAPACITY; i++) {
      if( flags[i] == 0.)
         continue;
      long j;
      for(j = i + 1; j < CELL_CAPACITY; j++) {
           if( flags[j] == 0.)
              continue;
           real dr[3];
           real dr2 = 0.;
           int d;
           for( d = 0; d < 3; d++ ) {
              dr[d] = r[3*j+d] - r[3*i+d];
              dr2 += dr[d] * dr[d];
           }
           if( dr2 > config.cutoff_radius_sq )
              continue;
           real invdr2 = 1. / dr2;
           config.count++;
           (*count_local)++;

           /* Lennard Jones interaction forces */
           real lj6 = sigma2 * invdr2;
           lj6 = lj6 * lj6 * lj6;
           real lj12 = lj6 * lj6;
           real lj12m6 = lj12 - lj6;
           real u_pot = epsilon24 * lj12m6;
           real factor = epsilon24 * (lj12 + lj12m6) * invdr2;
#ifndef NDEBUG
          if(isnan(factor)) {
             fprintf(stderr, "Diablock\n");
             fprintf(stderr, "i: %lu, j: %lu, dr2: %"PRIreal", invdr2: %"PRIreal"\n", i, j, dr2, invdr2);
             molecule_block_print_ascii(block, stderr);
          }
#endif
           /* Start of Critical Section*/
           upc_lock(lock_cell);
           for( d = 0; d < 3; d++ ) {
              F[3*i+d] -= factor * dr[d];
              F[3*j+d] -= F[3*i+d];
           }
           upc_unlock(lock_cell);
           /* End of Critical Section*/
           U_pot += u_pot;
      }
   }
   *E_local += U_pot;
   *U_Pot_local += U_pot;
}

void acquire_locks(upc_lock_t * lock_cell1, upc_lock_t * lock_cell2) {
           int lock1_flag = 0;
           int lock2_flag = 0;
           while(TRUE) {
              lock1_flag = upc_lock_attempt(lock_cell1);
```

```
            lock2_flag = upc_lock_attempt(lock_cell2);
            /* If not acquired both locks, release any one lock if acquired */
            if(lock1_flag && lock2_flag)
                  break;
            else {
                  if(lock1_flag)
                    upc_unlock(lock_cell1);
                  if(lock2_flag)
                    upc_unlock(lock_cell2);
                  }
        }
}
```

Figure 5.7(e): Routines for calculating forces acting on each molecule and integrating LJ potential

## 5.8 Post-Force Integration

The process of post-force integration is initiated by call to *psp_integrate_post* routine from main. It follows a hierarchy of calls as shown in figure 5.8(a). The process of post-force integration involves computation of velocities of all molecules inside phasespace and integrating kinetic energy of all molecules inside phase space. This process of post-force integration is performed in parallel fashion, such that all threads calculate velocities and integrate kinetic energy for the molecules residing in cells these threads have affinity. Finally UPC's reduction function is used to find the global integrated kinetic energy of the complete phasespace. The routines of *psp_integrate_postf* and *mc_integrate_postf* are shown in figures 5.8(b) and 5.8(c) respectively.

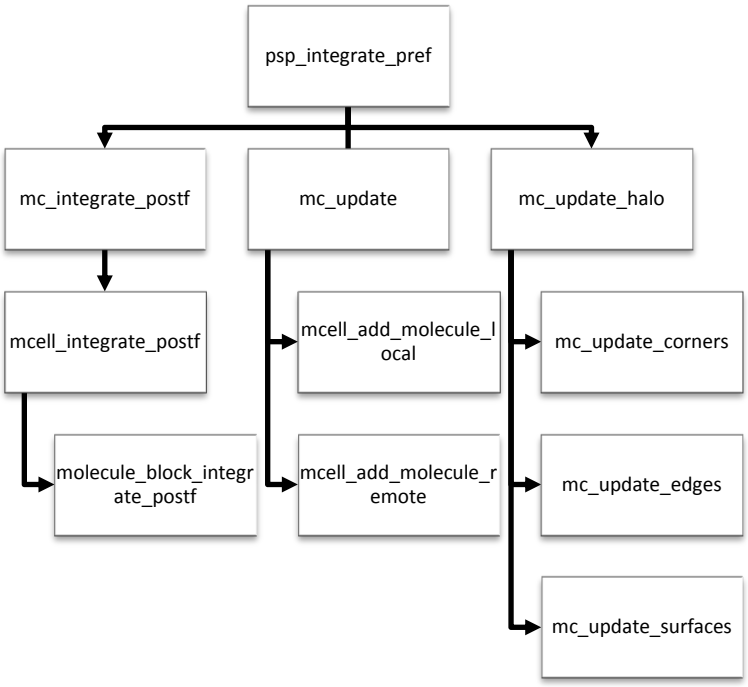


Figure 5.8(a): Post-force integration hierarchy of calls

After the global sum of integrated kinetic energy is computed, the *psp_integrate_postf* routine initiates the process of updating the phasespace. The process of updating phasespace involves removing the specific molecules from one cell and adding them to another cell. For this purpose each thread needs to acquire the locks on both source and destination cells. Once these specific molecules are moved from one cell to another, a thread having locked these cells releases the locks of these cells to allow other threads to acquire a lock on any of these cells if they need to do so. The routine *mc_update* can only

result in a deadlock if two threads have locked a source cell each and these cells are destination for each other. Hence both threads will be waiting to acquire lock on cell which is locked by the other thread. There is no such scenario observed in our case, however if in future such scenario occurs, it can be tacked by using the non-blocking routine for acquiring lock (*upc_lock_attempt*), as it is done in the inter-cell force calculation routines. The *mc_update* routine is shown in figure 5.8(d).

After process of updating phasespace is completed, it is required to assure that no thread is still in the process of updating phasespace. This is achieved by using barrier synchronization after a call to *mc_update* routine. Once all threads synchronize after updating the phase space in parallel fashion, a call to *mc_update_halo* routine is made inside *psp_integrate_postf* and with the completion of updating halo task, the complete process of post-force integration is finished.

```
void psp_integrate_postf(phasespace_t *psp, simulation_t *simulation, domain_t *domain, ensemble_t
*ensemble){
   mc_integrate_postf(psp->mc, simulation, ensemble);
   mc_update(psp->mc, domain);
   upc_barrier SYNC_POINT5;
   mc_update_halo(psp->mc);
}
```

Figure 5.8(b): psp_integrate_postf routine

```
shared [1] real e_kin_shared[THREADS];
shared real e_kin_final = 0;
void mc_integrate_postf(molecule_container_t *mc, simulation_t *simulation, ensemble_t *ensemble) {
            ensemble->E_kin = 0.0;
            e_kin_shared[MYTHREAD] = 0;
            long i, j, k;
   for(i = 1; i < mc->num_cells_per_dim[0] - 1; i++) {
      for(j = 1; j < mc->num_cells_per_dim[1] - 1; j++) {
         for(k = 1; k < mc->num_cells_per_dim[2] - 1; k++) {
            long cell_id;
            cell_id = get_cell_id( i, j, k ,mc);
            if(MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
               molecule_cell_t * cell = (molecule_cell_t *) &mc->cells[cell_id];
               real * e_kin_local = (real *) &e_kin_shared[MYTHREAD];
               mcell_integrate_postf(simulation, cell, e_kin_local);
            }

         }
      }
   }
   /********************************** SYNC_POINT4 **********************************/
   /* Below reduction routine provides barrier functionality as well as this collective routine
    * will only return when all reads and writes of data are complete */
   upc_all_reduceD(&e_kin_final, e_kin_shared, UPC_ADD, THREADS, 1, NULL, UPC_IN_MYSYNC |
                                       UPC_OUT_ALLSYNC);
   ensemble->E_kin = e_kin_final;
   ensemble->E = e_kin_final;
}
```

Figure 5.8(c): mc_integrate_postf routine

```
void mc_update(molecule_container_t *mc, domain_t *domain) {
   long i, j, k;
```

```
for(i = 1; i < mc->num_cells_per_dim[0] - 1; i++) {
  for(j = 1; j < mc->num_cells_per_dim[1] - 1; j++) {
    for(k = 1; k < mc->num_cells_per_dim[2] - 1; k++) {
      long cell_id;
      cell_id = get_cell_id( i, j, k ,mc);

      /* If the CELL has affinity to current thread */
      if (MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
          molecule_cell_t * cell = (molecule_cell_t *) &mc->cells[cell_id];
          /*Acquire lock on source cell so its not being modified while its being copied*/
          upc_lock(cell->lock_mycell);

          real *mblock = cell->data;
          real *flags = mblock;
          real *r = flags + CELL_CAPACITY;
          real *F = r + 3 * CELL_CAPACITY;
          real *v = F + 3 * CELL_CAPACITY;
          real *id = v + 6 * CELL_CAPACITY;
          real * size = id + CELL_CAPACITY;

          int i;
          for( i = 0; i < CELL_CAPACITY; i++) {
            if(flags[i] != 0.) {
              if( r[3*i+0] < 0.  ||  r[3*i+0] >= domain->L[0] || r[3*i+1] < 0.  ||  r[3*i+1] >=
                                    domain->L[1] || r[3*i+2] < 0.  ||  r[3*i+2] >= domain->L[2] )
                                                {
                /* save molecule data to struct */
                molecule_t m;
                m.id   = id[i];
                m.r[0] = r[3*i+0];
                m.r[1] = r[3*i+1];
                m.r[2] = r[3*i+2];
                m.v[0] = v[3*i+0];
                m.v[1] = v[3*i+1];
                m.v[2] = v[3*i+2];

                /* apply periodic boundary condtions */
                int d;
                for(d = 0; d < 3; d ++) {
                  if(m.r[d] < 0) m.r[d] += domain->L[d];
                  else if (m.r[d] >= domain->L[d]) m.r[d] -= domain->L[d];
                }

                /* Addition of molecule to shared space of other cell is surrounded by lock */
                /* calculate target cell and add molecule */
                long dest_cell_id = get_cell_index_from_coordinate(m.r[0], m.r[1], m.r[2], mc);

                if (MYTHREAD == upc_threadof(&mc->cells[dest_cell_id])) {
                  molecule_cell_t *dest_cell = (molecule_cell_t *) &mc->cells[dest_cell_id];
                  /* delete molecule */
                  flags[i] = 0.;
                   (*size)-=1.;
                  upc_lock(dest_cell->lock_mycell); /*lock dest cell so it can no be used as
                                                      source unless molecule is added to it */
                  mcell_add_molecule_local(&m, dest_cell);
```

```
                        upc_unlock(dest_cell->lock_mycell);
                    }
                    else {
                        shared [BLOCK_QUALIFIER] molecule_cell_t *dest_cell =
                                                                &mc->cells[dest_cell_id];
                    /* delete molecule */
                    flags[i] = 0.;
                    (*size)-=1.;
                    long dest_thread = upc_threadof(&mc->cells[dest_cell_id]);
                    upc_lock(dest_cell->lock_mycell); /*lock dest cell so it can no be used as source
                                                            unless molecule is added to it */
                    mcell_add_molecule_remote(&m, dest_cell);
                    upc_unlock(dest_cell->lock_mycell);
                    }
                }
            }
        }
        upc_unlock(cell->lock_mycell);
    }
  }
 }
}
```

Figure 5.8(c): mc_update routine

## 5.9 Calculation of Ensemble Values

Ensemble values are the core elements of our simulation. The result of our simulation is extracted from ensemble values. The computation of ensemble values is partially done in the force calculation and post-force integration routines, by integrating potential and kinetic energy for the molecules in phasespace. The other ensemble values which are required throughout the simulation are calculated by calling the routine *psp_calc_ensemble_values* from main function. This function finds the total number of molecules in phasespace in parallel manner and then the value of total number of molecules and integrated kinetic energy is used to find the value of temperature of phasespace. The routine for calculating the number of molecules in parallel fashion is shown in figure 5.9(a). Here each thread calculates the number of molecules in cells it has affinity to and finally a reduction call is made to calculate the global sum of number of molecules.

```
shared long final_count = 0;
shared [1] long count[THREADS]; // used to count number of molecules per thread

long mc_get_num_molecules(molecule_container_t *mc) {
  count[MYTHREAD] = 0;
  long i, j, k;
  for(i = 1; i < mc->num_cells_per_dim[0] - 1; i++) {
    for(j = 1; j < mc->num_cells_per_dim[1] - 1; j++) {
      for(k = 1; k < mc->num_cells_per_dim[2] - 1; k++) {
        long cell_id;
        long n;
        cell_id = get_cell_id( i, j, k ,mc);
        if(MYTHREAD == upc_threadof(&mc->cells[cell_id])) {
                molecule_cell_t * cell = (molecule_cell_t *) &mc->cells[cell_id];
                n = mcell_get_num_molecules(cell);
                count[MYTHREAD] += n;
```

```
        }
      }
    }
  }
  /********************************** SYNC_POINT6 **********************************/
  upc_all_reduceL(&final_count, count, UPC_ADD, THREADS, 1, NULL, UPC_IN_ALLSYNC |
                          UPC_OUT_ALLSYNC);
  return final_count;
}
```

Figure 5.9(a): *mc_get_num_molecules* routine

## 5.10 Synchronization in Parallel CMD code

There are various points of synchronization in our parallel version of CMD code. These synchronization points are assigned the integer value and are defined as global constants in one of the header files, as shown in figure 5.10(a). These synchronization points are explained below.

```
#ifndef SYNC_H
#define SYNC_H

/* Define integer values for various barrier synchronization points */
#define SYNC_POINT0 0
#define SYNC_POINT1 1
#define SYNC_POINT2 2
#define SYNC_POINT3 3
#define SYNC_POINT4 4
#define SYNC_POINT5 5
#define SYNC_POINT6 6
#define SYNC_POINT7 7
#define SYNC_POINT8 8
#define SYNC_POINT9 9
#define SYNC_POINT10 10

#endif /* SYNC_H */
```

Figure 5.10(a): Definition of synchronization constants

**Lock per Cell**
Location: Inside the structure of cell
Definition: Lock of each cell is used to modify forces of and adding molecules to neighbor cells atomically.

**SYNC_POINT0**
Location: Before main loop (after printing stats and before writing ASCII Output)

**SYNC_POINT1**
Location: After pre-force integration step inside main loop.
Definition: Makes sure that every thread has calculated values of r and v of its responsible molecule blocks, values of r are being later used by neighbor cells to calculate distance between molecules for force calculation.

**SYNC_POINT2**
Location: At the end inside Update Halo routine.
Definition: Molecule blocks are being copied from one cell to another by thread which has affinity to destination cell. This barrier makes sure that NO cell of a thread is being currently used as source cell

before any thread starts manipulating its cells.

**SYNC_POINT3**
Location: At the end inside mc_calc_forces
Definition: Need synchronization at this point as every thread is manipulating Forces of molecules of neighboring cells and neighbor cell calculates values of v and e_kin of its molecules depending on value of forces of its molecules.

**SYNC_POINT4**
Location: At the end inside mc_integrate_postf
Definition: Makes sure that every thread has calculated velocity of molecules of its cells before update routine is called

**SYNC_POINT5**
Location: Inside psp_integrate_postf routine after calling mc_update routine
Definition: Makes sure that molecules have been moved between cells (inside mc_update routine) before copying cells (inside mc_update_halo routine).

**SYNC_POINT6**
Location: At the end inside mc_get_num_molecules
Definition: Makes sure that every thread has calculated its number of molecules before calculating global sum of molecules.

**SYNC_POINT7**
Location: Inside Main Loop before writing ASCII Output
Definition: Makes sure that all threads have done calculation before THREAD 0 starts writing output on file

**SYNC_POINT8**
Location: Inside Main Loop after writing ASCII Output
Definition: Makes that no other thread start next iteration before THREAD 0 stops writing to file

**SYNC_POINT9**
Location: After main loop (after clearing Halo and before starting writing ASCII Output)

**SYNC_POINT10**
Location: After main loop (after writing ASCII Output and before printing stats)

# 6 Benchmarks

## 6.1 Introduction

We have executed our parallel CMD code on two clusters, Nehalem and Cray. The specifications of these clusters are important to know for understanding the benchmark results. Figure 6.1 shows the specifications of these platforms.

| NEC Nehalem Cluster | Cray Hermit Cluster |
|---|---|
| 700 dual CPU compute nodes: 2x Intel Xeon X5560 "Gainestown" (5000 Sequence specifications)<br>▪ 4 cores, 8 threads<br>▪ 2.80 GHz (3.20 Ghz max. Turbo frequency)<br>▪ 8MB L3 Cache<br>▪ 1333 MHz Memory Interface, 6.4 GT/s QPI<br><br>Network: InfiniBand Double Data Rate switches for interconnect: Voltaire Grid Director 4036 with 36 QDR (40Gbps) ports (6 backbone switches) | 3552 compute nodes<br>▪ 2x AMD Opteron(tm) 6276 (Interlagos) processors with 16 Cores @ 2.3 GHz (with TurboCore up to 3.3 GHz)<br>▪ 32MB L2+L3 Cache, 16MB L3 Cache<br>▪ HyperTransport HT3, 6.4GT/s=102.4 GB/s<br><br>Network: High Speed Network CRAY Gemini |

Figure 6.1: Specifications of benchmark platforms

In the following sections we will discuss different benchmarks carried out on clusters just explained.

## 6.2 Speed up on Different Clusters

The first benchmark carried out after porting the CMD application to Unified Parallel C is shown in figure 6.2. This benchmark does not utilize Newton's 3rd law of motion; hence force computation is done twice for each pair of molecules in the phasespace fulfilling the cutoff radius threshold. On the other hand it relieves with necessary synchronization required every time the force of neighboring cell's molecule is to be modified. Our CMD application is executed with two test cases of varying number of molecules, N=500 and N=5000. A very good speed up is observed on both Nehalem and Cray clusters after porting the CMD to Unified Parallel C utilizing new data structures and algorithms. However a sudden dip is observed in the speed up of application when executed on Nehalem cluster with more than 8 cores. This happens because the inter-node communication comes into action on Nehalem cluster when going beyond 8 cores. When inter-node communication is involved, the communication overhead surpasses the computation time, resulting in poor speed up of application.

## 6.3 Slow Down Factor with Synchronization Variants

Although the use of Newton's 3rd law of motion reduces the computation effort in our simulation, it raises the urgency of synchronization. While porting the CMD to Unified Parallel C, we have employed two strategies for synchronization among threads, critical section synchronization and point to point synchronization. In critical section synchronization, each thread must acquire the lock of critical region where force of a neighbor cell's molecule is modified. Hence, while one thread is modifying the force of molecule of one of its neighbor cell's molecule, the other threads can only compute or modify the forces on molecules of its cell acting due to other molecules in the same cell. Hence only one thread can modify force of its neighbor cell's molecule at a time even though different threads might need to access different neighbor cells. This results in excessive synchronization as a thread might need to modify force of a molecule in one of its neighbor cells that is not being modified by other thread yet it needs to wait for other thread to release the lock. For countering the superfluous synchronization

introduced by critical section synchronization mechanism, we employed another strategy by introducing a new data structure for a cell where each cell has its own lock. By using a lock per cell, we employed point to point synchronization mechanism where a thread only locks the cell it needs to modify the forces of molecules of this cell. Hence multiple threads can modify the forces of molecules of their neighbor cells simultaneously if they don't have a neighbor cell in common. When two threads need to access a similar cell, they must wait for each other to acquire lock on this cell.

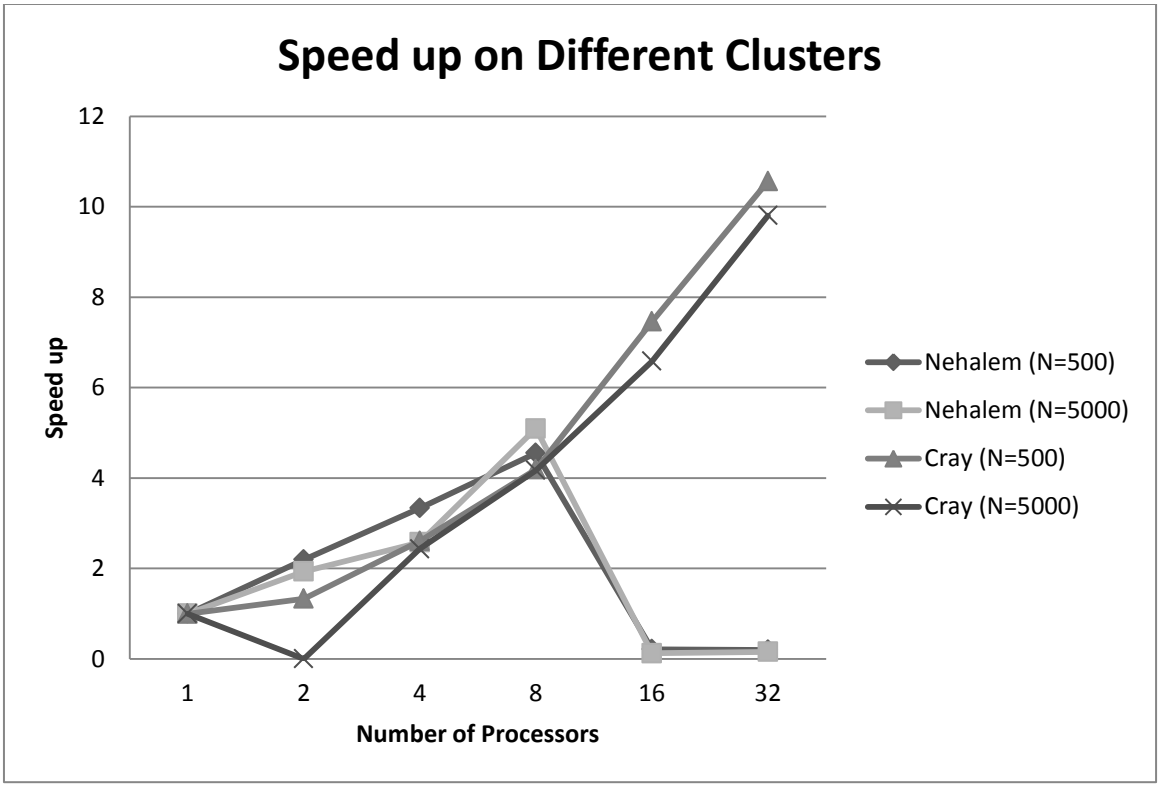


Figure 6.2: Speed up on different clusters

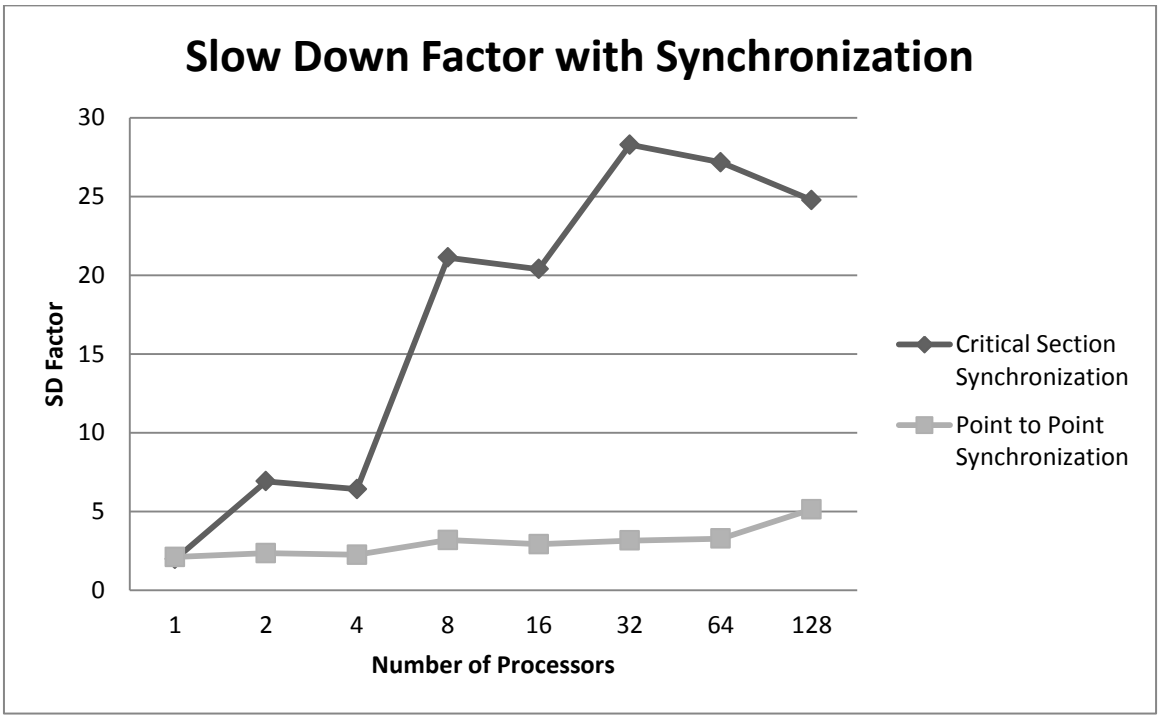


Figure 6.3: Slow Down Factor Due to Synchronization

Figure 6.3 shows the benchmark which demonstrates the slowdown in our CMD application due to the introduction of synchronization semantics using two techniques just discussed. It can be clearly seen in the benchmark that the introduction of critical section synchronization technique results in our CMD application to be 30 times slower as compared to the non-synchronization version of CMD. However, intelligent point to point synchronization results in a maximum slowdown factor of 5. Hence our CMD code implements the point to point synchronization in its final version.

## 6.4 Execution Time with different Cells Distribution

While porting our CMD application to UPC, we exercised two approaches for the distribution of molecule cells among threads, round-robin and spatially coherent cells distributions. In the round-robin approach, cells are distributed among threads sequentially, as shown in figure 6.4(a). Whereas in spatially coherent cells distribution, spatially coherent cells are allocated to the same thread, as shown in figure 6.4(b).

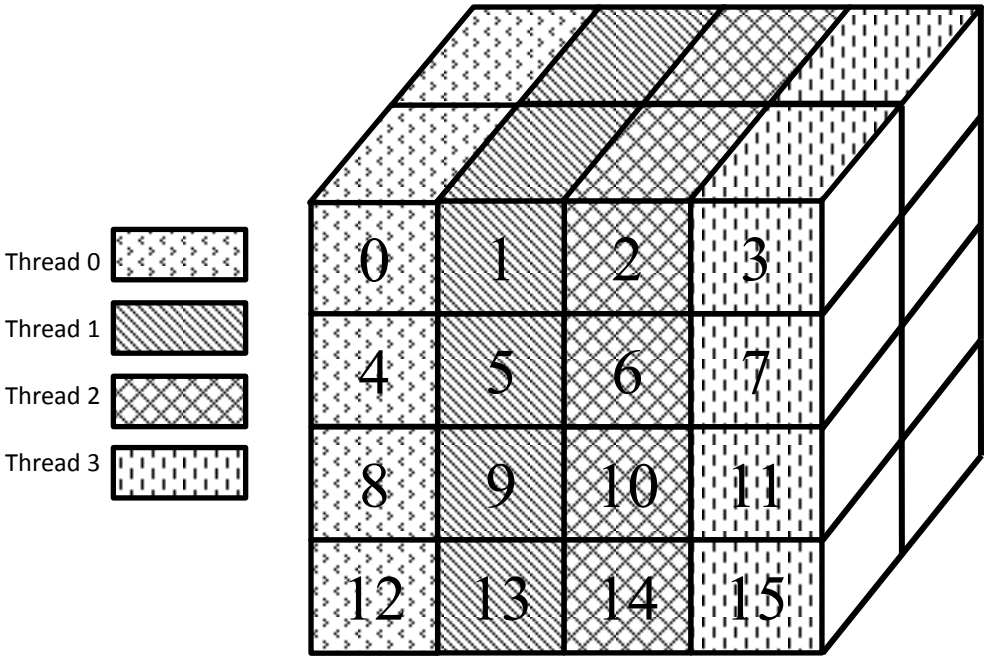


Figure 6.4(a): Round Robin cells distribution

Figures 6.4(c), 6.4(d) and 6.4(d) show the benchmarks of execution time of CMD application on Nehalem cluster with total number of molecules in phase space equals to 1800, 3000 and 27000 respectively. It can be clearly seen in all three benchmarks that the distribution of cells in a spatially coherent manner results in significantly less execution time of our application compared to round robin distribution of cells among threads. For the benchmarks shown in figures 6.4(c) and 6.4(d), the application is executed faster even after going beyond one node on Nehalem where inter-node communication comes into action. This is because spatially coherent cells distribution results in very less remote data access as compared to round robin cells distribution. Hence the locality of data is exploited more effectively when cells are distributed in spatially coherent manner. For very large number of molecules, in figure 6.4(e), spatially coherent cells distribution still performs better as compared to round-robin cells distribution. Recall that the molecules cells are distributed among threads in a three dimensional spatial coherence, blockDimx*blockDimy*blockDimz (a 3x3x3 cube of molecule cells belongs to the same thread). As we can see in figure 6.4(e), inter-node communication has an impact on performance with very large number of molecules even with distribution of cells in spatially coherent manner. This is because the number of cells is changed with number of molecules, and the relationship between number of cells and number of threads establishes how well the cells can be spatially decomposed. The ratio of surface to volume of the phasespace plays a significant role in scaling.

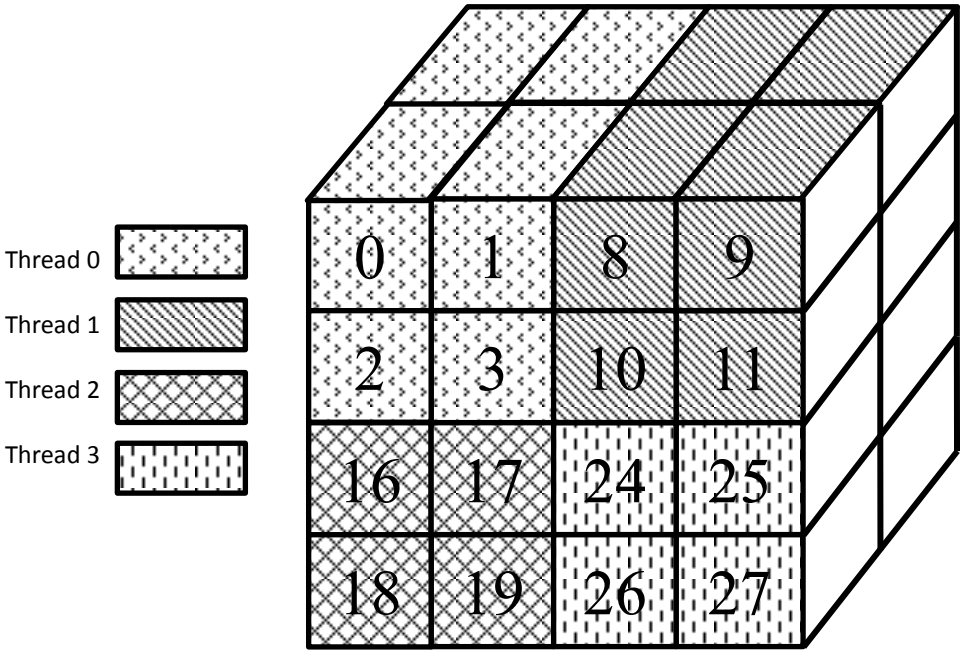


Figure 6.4(b): Spatially coherent cells distribution

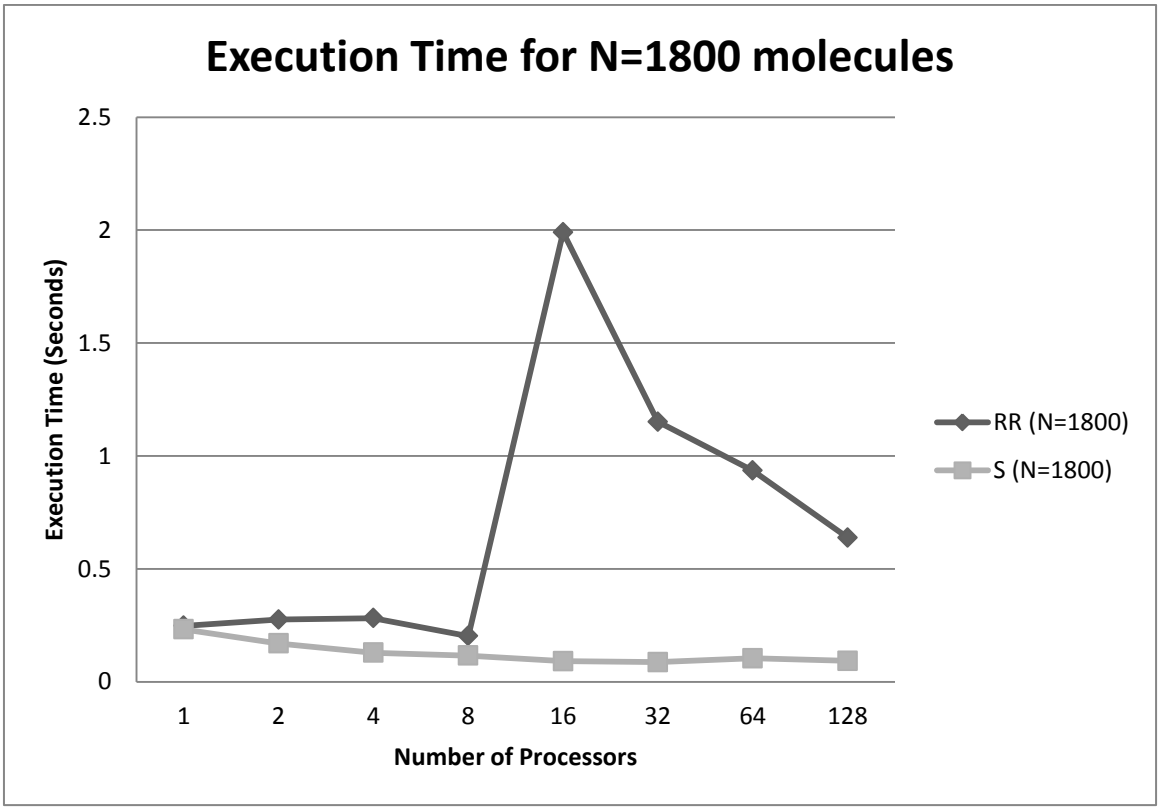


Figure 6.4(c): Execution Time for different cell distributions and N=1800 molecules

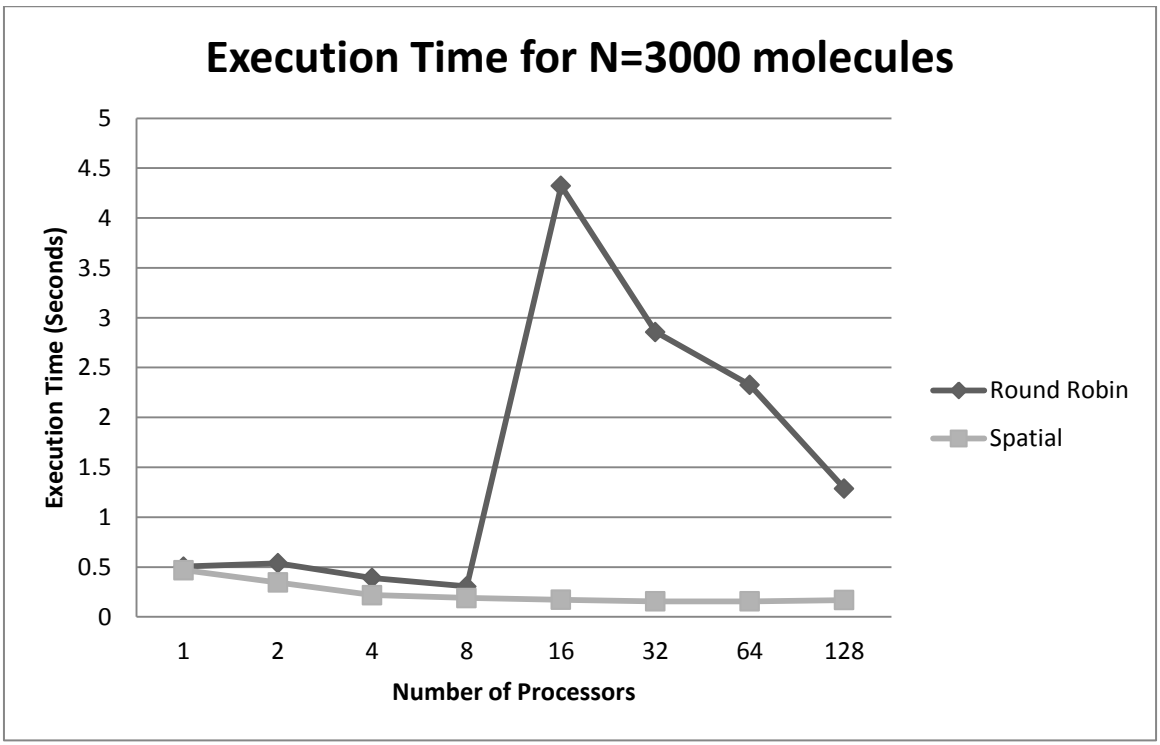


Figure 6.4(d): Execution Time for different cell distributions and N=3000 molecules

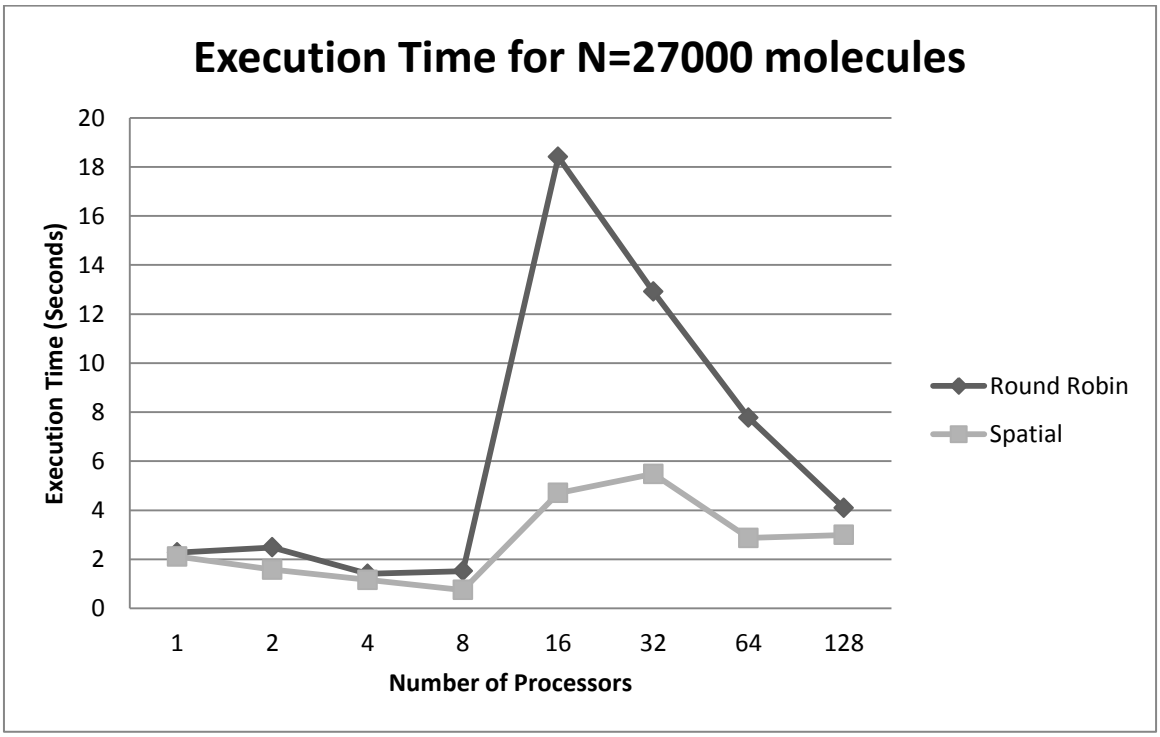


Figure 6.4(e): Execution Time for different cell distributions and N=27000 molecules

## 6.5 Execution Time with UPC pointer variants

In earlier version of our ported CMD application, each thread accessed its neighbor cells using private pointers to shared region in the inter-cell force calculation routine whereas it accessed only the source cell using local pointer after checking the affinity of source cell. In such case there is the possibility that a cell and its neighbor cell are both located in the local portion of the shared address space. As for each cell, there are 13 neighbor cells, we implemented manual pointer optimizations where the affinity of neighbor cell is also checked and it is accessed using local pointer when it also resides in the local portion of a thread in the shared space. After executing our CMD application with manual pointer optimizations in place, we saw a significant reduction in execution time for all tested cases of varying number of molecules. This is shown in figure 6.4. For all three test cases shown below, our application executed 10 times faster than the earlier version which did not utilize manual pointer optimizations for accessing neighbor cells in force calculation routine. This shows that the UPC compilers are still not mature enough to detect the data locality automatically and accessing the data using local pointers where it resides in local portion of shared space for a thread.

- **Test case 1 -- N=500 molecules, NumThreads = 1**
  - RR Distribution with shared pointers = 0.27666
  - RR Distribution with local pointers = 0.027639 **(10x faster**)
- **Test case 2 -- N=3000 molecules, NumThreads = 1**
  - RR Distribution with shared pointers = 0.437530
  - RR Distribution with local pointers = 0.042233 **(10x faster)**
- **Test case 3 -- N=27000 molecules, NumThreads = 1**
  - RR Distribution with shared pointers = 1.946293
  - RR Distribution with local pointers = 0.195274 **(10x faster)**

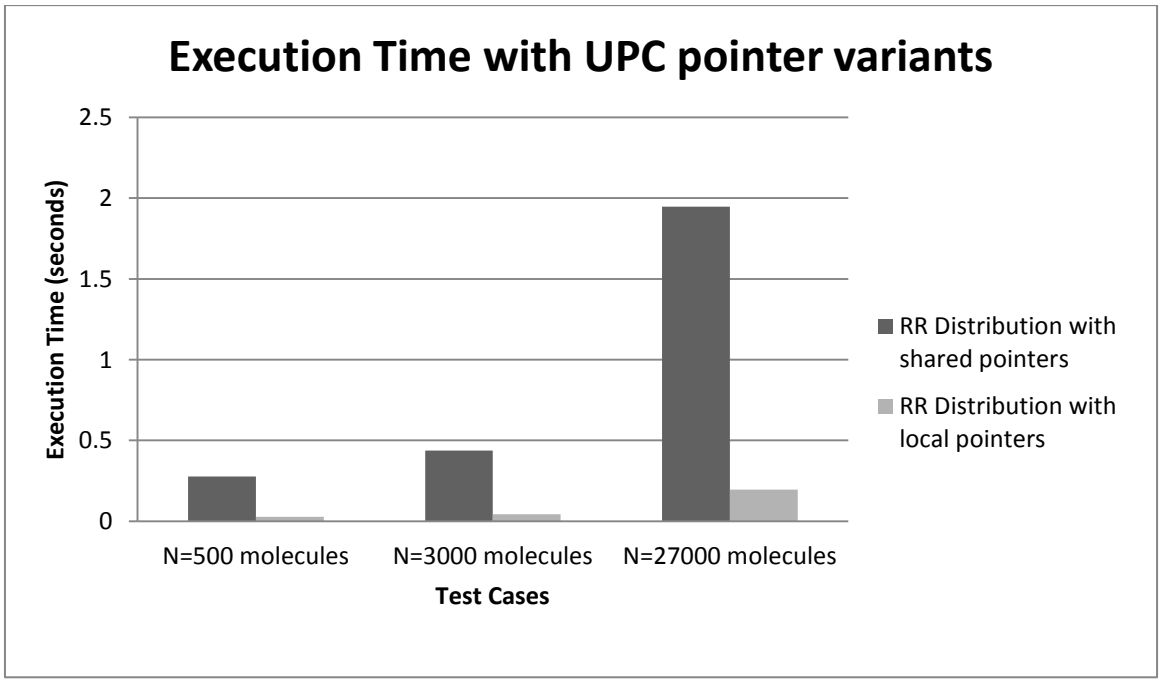


Figure 6.5: Execution Time with manual pointer optimizations

In the next chapter we will discuss the performance bottlenecks of UPC we observed in this chapter and will suggest possible solutions.

# 7 Bottlenecks and Solutions

We observed two major performance bottlenecks while porting our application to UPC. These are discussed in this chapter with the possible solutions.

## 7.1 Source to Source Translation Issues

The most critical performance issue we observed while porting our CMD application to UPC is the source-to-source translation issue. We have observed that UPC compilers are still not aided with few major performance optimizations. UPC compilers require a programmer to manually cast a shared pointer to a private pointer when the data pointed by the shared pointer resides in the local portion of shared space of this thread. UPC compilers are to date not able to detect the data locality perfunctory and automatically accessing the local portion of shared space for a thread using a local pointer. This can result in severe performance degradation if UPC code is not optimized manually for pointers, as shown in the benchmark in figure 6.5. It is also recommended to inform the UPC compiler explicitly if pointers used in program do not alias, such that they do not point to same memory location [1]. It is recommended to analyze the program and use the ANSCI C99 *restrict* keyword to inform the UPC compiler if pointers do not alias. This can result in cache optimization by not loading a value repetitively due to compiler being unaware of the pointer aliasing [1]. By using the *restrict* keyword, the UPC source-to-source translator aids the back end ANSI C compiler with the information about pointer aliasing. Hence it is recommended to do manual optimizations in UPC code unless the UPC compilers become mature enough to do these optimizations automatically.

## 7.2 Excessive Synchronization

Although the use of synchronization in programs assures the correct data coordination among threads, it is challenging to have an optimal synchronization in a parallel program. An inefficient use of synchronization constructs can result in unnecessary communication among threads and would result in wastage of machine cycles. This becomes even more important when the UPC's synchronization constructs are employed in a program as they incur an implicit overhead due their global shared scope. This is the reason that the other major performance issue in a UPC program occurs due to inefficient use of the synchronization constructs. We have observed that unnecessary use of barrier synchronization or having the scope of a lock for large shared data can result in performance degradation. To achieve better performance of UPC programs it is suggested to only use barrier synchronization when it is really necessary for a thread to wait for all other threads. This happens when a thread needs to enter the next phase in a program and requires accessing a shared data which can be modified by all other threads or majority of threads. It is recommended to use non-blocking barrier synchronization in such scenarios by designing your program in a way that a thread can do local computation while waiting for other threads to reach at a point in program. It is also suggested to optimize the scope of lock for shared data, such that it should result in as less collision among threads to acquire a lock as possible. This can be done by designating a lock for small data set as compared to large data set. On the other hand allocating so many locks in program can also result in data access overhead. Therefore an optimal value for number of locks should be selected.

# 8 Conclusion

UPC would be a very productive language if in future UPC compilers aid the necessary optimizations which have significant impact on the performance of a program. UPC provides ease of programming due to its global address space abstraction and powerful parallel programming mechanisms with only few extensions to ANSI C language [4]. UPC provides support for both shared and distributed memory architectures and hence it can be used for hybrid parallelism conventionally provided by MPI and OpenMP combination. However UPC compilers are still not matured enough to encompass some major optimizations which a programmer need to do manually in program. Currently it is essential for a programmer to pay special attention to all types of pitfalls which seriously attenuates the productivity and (if a programmer is unable to recognize all the issues) the performance. Therefore it is still not ready for the production code.

While porting our CMD code we exercised different strategies for distribution of data among threads, algorithms and synchronization mechanism. The round robin cells distribution is the default data distribution strategy and hence it is essential to hunt for better data distribution methods. The spatial cells distribution strategy is one such initiative of achieving better data (or cells in our case) distribution. We found out that distribution of molecules cells among threads in spatially coherence manner reduces the remote accesses significantly and outperforms the round-robin cells distribution strategy. The spatially coherence distribution strategy also performed exceptionally well when our application is executed on more than one node and internode communication took place, unlike round robin distribution. In algorithms, we found out that designing locality aware algorithms for the compute intensive routines of force calculation with manual pointer optimizations results in our application to be executed really faster as UPC compilers were unable to detect the locality of data and access it using local pointers automatically in the earlier version which resulted in high execution time. Lastly, the intelligent point to point synchronization strategy has very significant impact on the execution time of our simulation, compared to earlier used critical section synchronization which had very high slow down factor which was directly proportional to number of nodes. It is recommended to utilize the synchronization mechanisms efficiently and not to exercise the code with excessive synchronization. For example, programmers should devise the strategy of using non-blocking barrier synchronization to overlap the computation and communication unlike blocking barriers which results in inefficient use of machine cycles.

In future, we intend to investigate the larger simulations and more adaptive spatial decomposition methods.

# 9 References


[1] Cristian Coarfa , Yuri Dotsenko , John Mellor-Crummey , François Cantonnet , Tarek El-Ghazawi , Ashrujit Mohanti , Yiyi Yao , Daniel Chavarría-Miranda, An evaluation of global address space languages: co-array fortran and unified parallel C, Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, June 15-17, 2005, Chicago, IL, USA

[2] Parry Husbands , Costin Iancu , Katherine Yelick, A performance analysis of the Berkeley UPC compiler, Proceedings of the 17th annual international conference on Supercomputing, June 23-26, 2003, San Francisco, CA, USA

[3] François Cantonnet , Yiyi Yao , Smita Annareddy , Ahmed S. Mohamed , Tarek A. El-Ghazawi, Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture, Proceedings of the 17th International Symposium on Parallel and Distributed Processing, p.274.2, April 22-26, 2003

[4] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In IPDPS, 2004.

[5] Tarek El-Ghazawi , William Carlson , Thomas Sterling , Katherine Yelick, UPC: Distributed Shared Memory Programming, Wiley-Interscience, 2003

[6] Katherine Yelick , Dan Bonachea , Wei-Yu Chen , Phillip Colella , Kaushik Datta , Jason Duell , Susan L. Graham , Paul Hargrove , Paul Hilfinger , Parry Husbands , Costin Iancu , Amir Kamil , Rajesh Nishtala , Jimmy Su , Michael Welcome , Tong Wen, Productivity and performance using partitioned global address space languages, Proceedings of the 2007 international workshop on Parallel symbolic computation, July 27-28, 2007, London, Ontario, Canada

[7] Z. Zhang and S. Seidel. Benchmark Measurements for Current UPC Platforms. In Proceedings of IPDPS'05, 19th IEEE International Parallel and Distributed Processing Symposium, Apr. 2005.

[8] Tarek El-Ghazawi , Francois Cantonnet, UPC performance and potential: a NPB experimental study, Proceedings of the 2002 ACM/IEEE conference on Supercomputing, p.1-26, November 16, 2002, Baltimore, Maryland

[9] M.P. Allen, Introduction to molecular dynamics simulation, in: Computational Soft Matter-From Synthetic Polymers to Proteins, NIC Series, vol.23, John von Neumann Institute for Computing, 2004.

[10] T.W Clark, R.v Hanxleden, J.A McCammon, L.R Scott
Parallelizing molecular dynamics using spatial decomposition
Proceedings, Scalable High-Performance Computing Conference (1994), pp. 95–102

[11] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high performance computing. ACM Computing Surveys, 26(4):345{420, December 1994. http://www.acm.org/pubs/toc/Abstracts/03600300/197406.html.

[12] H. El-Rewini, M. Abd-El-Barr
Advanced Computer Architecture and Parallel Processing
Wiley (2005) ISBN 978-0-471-46740-

[13] Domenico Laforenza
Parallel computer architectures: state of the art and trends
CNUCE-Istituto del CNR, Reparto Calcolo Parallelo, Via S. Maria, 36, 1-56100 Pisa, Italy
Received September 12, 1990/Accepted November 13, 1990

[14] Paul Graham , OpenMP A Parallel Programming Model for Shared Memory Architectures Version 1.1 March 1999
Edinburgh Parallel Computing Centre, The University of Edinburgh

[15] David J. Lilja, Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons, ACM Computing Surveys (CSUR), v.25 n.3, p.303-338, Sept. 1993

[16] Barney B. Introduction to parallel computing. Lawrence Livermore National Laboratory, 2010

[17] Jelica Protic , Milo Tomasevic , Veljko Milutinovic, Distributed Shared Memory: Concepts and Systems, IEEE Parallel & Distributed Technology: Systems & Technology, v.4 n.2, p.63-79, June 1996

[18] Leonardo Dagum , Ramesh Menon, OpenMP: An Industry-Standard API for Shared-Memory Programming, IEEE Computational Science & Engineering, v.5 n.1, p.46-55, January 1998

[19] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specification v1.1.1, October 2003

[20] MPI: A Message-Passing Interface Standard Version 2.2
Message Passing Interface Forum
September 4, 2009

[21] MPI (Message Passing Interface), High Performance Computing Virtual Laboratory Canada
http://www.hpcvl.org/faqs/programming/mpi-message-passing-interface

[22] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication Optimizations for Fine-grained UPC Applications. In Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, 2005.

[23] G. Taubenfeld. Shared memory synchronization. Bulletin of the European Association for Theoretical Computer Science, 96:80–, October 2008. Columns: Distributions

[24] Synchronization Barriers – Microsoft Dev Center
Dev Center - Desktop > Docs > Windows Development Reference > System Services > Synchronization > About Synchronization > Synchronization Barriers
http://msdn.microsoft.com/en-us/library/windows/desktop/hh706897(v=vs.85).aspx

# Acknowledgment

I would like to express my gratitude to all those who made it possible for me to complete this thesis and enabled me to achieve my goal.

First of all, I am very indebted to my supervisor Dr. Colin Glass whose help, precious suggestions, comprehension, perspicacity and encouragement helped me in all the times of thesis. I am also really thankful to Christoph Niethammer, who really encouraged me for all the efforts I made and helped me with his invaluable knowledge on subject.

I would also like to thank my supervisors Prof. Dr.-Ing. Sven Simon and Dipl.-Inf. Steffen Kieß for providing me with the opportunity to work with them and their entrustment on me for this thesis.

Lastly but importantly, I am really very thankful to Prof. Dr. Rainer Keller who really encouraged me for my interest in Parallel Systems and providing me with the prospect of working with him in this field.

## Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, 17. Oktober 2012