INSTITUTFÜRPARALLELE UND VERTEILTE SYSTEME

ABTEILUNG BILDVERSTEHEN

UNIVERSITÄT STUTTGART

UNIVERSITÄTSTRASSE 38

D-70569 STUTTGART

HIGH PERFORMANCE COMPUTING CENTER STUTTGART (HLRS)

UNIVERSITÄT STUTTGART

NOBELSTRASSE 19

D-70569 STUTTGART

Master Thesis Nr. 3215

Hybrid Parallel Computing beyond MPI &OpenMP - Introducing PGAS &StarSs

Muhammad Wahaj Sethi

Studiengang:	M.Sc. in Information Technology
Prüfer:	Prof. Dr. rer. nat. habil. Paul Levi
Betreuer:	Dr. rer. nat. habil. Viktor Avruti
	Dr. José Gracia
Begonnen am:	04.07.2011
Beendet am:	09.01.2011
CR-Klassifikation:	D.1.3, I.2.6

Abstract

High-performance architectures are becoming more and more complex with the passage of time. These large scale, heterogeneous architectures and multi-core system are difficult to program. New programming models are required to make expression of parallelism easier, while keeping productivity of the developer higher.

Partition Global Address-space (PGAS) languages such as UPC appeared to augment developer's productivity for distributed memory systems. UPC provides a simpler, shared memory-like model with a user control over data layout. But it is developer's responsibility to take care of the data locality, by using appropriate data layouts.

SMPSs/StarSs programming model tries to simplify the parallel programming on multi-core architectures. It offers task level parallelism, where dependencies among the tasks are determined at the run time. In addition, runtime take cares of the data locality, while scheduling tasks. Hence, providing two-folds improvement in productivity; first, saving developer's time by using automatic dependency detection, instead of hard coding them. Second, save cache optimization time, as runtime take cares of data locality.

The purpose of this thesis is to use the PGAS programming model e.g. UPC for different nodes with the shared memory task based parallelization model i.e. StarSs to take the advantage of the multi core systems and contrast this approach to the legacy MPI and OpenMP combination. Performance as well as programmability is considered in the evaluation.

The combination UPC + SMPSs, results in approximately the same execution time as MPI and OpenMP. The current lack of features such as multi-dimensional data distribution or virtual topologies in UPC, make the hybrid UPC + SMPSs/StarSs programming model less programmable than MPI + OpenMP for the application studied in this thesis.

Acknowledgment

I would like to give thanks to Dr. José Gracia whose guidance, help and encouragement make this thesis possible. I would also like to thanks Christoph Niethammer for providing useful debugging tips.

Muhammad Wahaj Sethi

Table of Contents

1 Introduction 1

2 Parallel programming models and paradigms 3

2.1 Levels of parallelism 3

2.1.1 Bit level parallelism 3

2.1.2 Instruction-level parallelism 4

2.1.3 Data parallelism 4

2.1.4 Task parallelism 5

2.2 Programming models 5

2.2.1 Message passing model 5

2.2.2 Shared memory model 6

2.2.3 Data parallel model 6

2.2.4 Distributed shared memory model 7

3 SMP superscalar (SMPSs/StarSs) 9

3.1 Programming model **10**

3.2Scheduling **12**

4 Unified Parallel C 15

4.1 Programming Model **15**

4.2 Shared and Private Variables **16**

4.3 Shared Arrays **18**

4.4 UPC Pointers **21**

5 Implementation details and results 23

5.1 StarSs **24**

5.2 UPC **30**

5.3 StarSs + UPC **33**

5.4 MPI **37**

5.5 OpenMP **42**

5.6 MPI + OpenMP **44**
6 Comparative performance **47**
7 Discussion **51**
8 Bibliography **53**
Appendix A **54**
A.1. Graph data tables **54**
Appendix B **66**
B.1 StarSs row-wise matrix multiplication code **66**
B.2 StarSs block-wise matrix multiplication code **72**
B.3 StarSs + UPC matrix multiplication code **80**
B.3.1 starSs.h 80
B.3.2. starSs.c 81
B.3.3. hybrid.h 85
B.3.4. hybrid.upc 86

List of Figures

Figure 2.1: Pipeline of RISC processor 4
Figure 2.2: Data dependency example 4
Figure 2.3: Address space and execution in parallel programming models [2] 8
Figure 3.1: Wait and barrier directives example code 12
Figure 3.2: SMPSs run time environment [4] 14
Figure 4.1: UPC memory and programming model [4] 16
Figure 4.2: Memory layout of variables[4] 18
Figure 4.3: Array distribution among threads for block size 4 [4] 19
Figure 4.4: Array distribution among threads for block size 3[4] 20
Figure 4.5: Contiguous array distribution among threads [4] 20
Figure 4.6: Two dimensional array distribution among threads[4] 21
Figure 4.7: Pointer classes[4] 21
Figure 4.8: Pointer classes memory layout and referencing space [4] 22
Figure 5.1: Row block distribution of matrices 25
Figure 5.2: Dependency tree for row block distribution 25
Figure 5.3: Block wise data distribution of matrices 26
Figure 5.4: Dependency tree for block wise data distribution 27
Figure 5.5: Matrix layout in memory 31
Figure 5.6: Matrix square block data distribution 33
Figure 5.7: Data blocks with respective owner process Cartesian coordinates 37
Figure 5.8: Row and column grouping of data blocks 38
Figure 5.9: Non blocking send/recv messages for process with Cartessian coordinates (0, 1) 39
Figure 5.10: Result matrix (C) gathering scheme. 40

List of Graphs

Graph 5.1: Row block distribution 28
Graph 5.2: Square block distribution 28
Graph 5.3: Row wise and block wise data distribution 29
Graph 5.4: Scalability plot (square block distribution) 29
Graph 5.5: Scalability plot (row block distribution) 30
Graph 5.6: UPC matrix multiplication (using block wise distribution) on one node 32
Graph 5.7: Scalability plot(using block wise distribution) on multiple nodes 32
Graph 5.8: Scalability plot (using row wise distribution) on one node 33
Graph 5.9: Improvement in performance by increasing number of cores - square block distribution 36
Graph 5.10: Scalability plot (multiple nodes) - square block distribution. 37
Graph 5.11: Execution time for various matrix sizes (Non-blocking send/recv) 40
Graph 5.12: Scalability plot (Non-blocking send/recv) 41
Graph 5.13:Execution time for various matrix sizes (all gather) 41
Graph 5.14: Scalability plot (all gather) 42
Graph 5.15: Comparison for matrix size 2048X2048 42
Graph 5.16: Matrix execution time for various cores 44
Graph 5.17: Scalability plot 44
Graph 5.18: Execution time for various matrix sizes (all gather) 45
Graph 5.19: Scalability plot 46
Graph 6.1: Shared memory comparison for 2048X2048 47
Graph 6.2: Shared memory comparison for 1024X1024 48
Graph 6.3: Shared memory comparison for 512X512 48
Graph 6.4: Hybrid (shared + distributed) memory comparison for 2048X2048 matrix 49

Graph 6.5: Hybrid (shared + distributed) memory comparison for 1024X1024 matrix 49
Graph 6.6: Hybrid (shared + distributed) memory comparison for 512X512 matrix 50

1 Introduction

Traditionally, HPC applications are developed using MPI [1] for distributed memory and OpenMP [2] for shared memory. Writing codes using MPI for message communication among nodes is a time consuming task. As user has to restructure its sequential code, data stored locally in the sequential code may lie on the remote node in the MPI version. Messages need to be sent to fetch remote data. OpenMP is the *de facto* way of exploiting shared memory architectures. But it suffers from the problem of scalability for very large number of cores.

Up till now for HPC applications, mostly MPI is used for distributed parallel computing. Its programming model is quite complex which makes it hard to write and maintain code. PGAS (Partition global address space) languages such as Unified Parallel C (UPC) [3] provide a simpler programming model which is easy to understand. Thus providing an ideal candidate for easy to write and maintain code. Until recently, PGAS compilers and runtime are inefficient. A situation - destined to change in the coming years.

UPC reduces the development time, by improving programmability. It brings in the ease of accessing memory location, in the shared memory systems to the distributed memory systems i.e. remote memory location can be referenced using normal assignment operators. In contrast to MPI, where we have to send messages for exchanging data, in UPC data can be exchanged by using normal assignment operator. Hence for data exchange, less number of lines of code needs to be written. Or we can say that, in UPC programs code overhead is significantly reduced as compared to the MPI programs.

New advancements in chip fabrication technology have allowed putting couple of billions of transistors on the chip. Several complex issues have discouraged, design of the complex processors. For a way out, people start increasing numbers of cores present on the chip. Multi-core chips are readily available in the market and in future we may have chips with 1000 of cores (many-core). Our hardware is developing quite rapidly as compared to programming tools. The StarSs parallel programming model is an effort to meet new hardware requirements.

Most of the applications are compromised of tasks, where every task implements a specific functionality and its output might be used by another tasks to produce final results. SMPSs/StarSs [4] provides a way for scheduling these tasks in the optimal way while taking care of the data locality.

In order to evaluate the performance, SUMMA (Scalable universal matrix multiplication algorithm) algorithm is implemented using pure UPC, pure MPI, UPC + StarSs combination and MPI + OpenMP combination. Their execution times are compared and discussed. In addition to the performance analysis, UPC and StarSs are also analyzed for programmability.

Work done in the thesis is organized as fallow: Chapter 2 discusses different parallel programming models. Chapter 3 conveys information about the SMP superscalar (SMPSs/StarSs) programming model. Overview of the Unified Parallel C (UPC) can be seen in Chapter 4. Chapter 5 describes implementation details and present results. Performance among different versions is compared in Chapter 6. Chapter 7 provides a general discussion on StarSs and UPC.

2 Parallel programming models and paradigms

In 1980's decade, it was widely believed that the computer performance was best improved by making faster and more efficient processor. The concept of the parallel processing challenged this belief. In the parallel processing two or more computers are linked together to solve a computation intensive problem. From 1990 onwards people start building the super computer (so called clusters) by making network of simple/readily available processors instead of using stand alone high performance massively parallel processors. This trend is further enhanced by increasing high availability and low price of network equipment for connecting computers. Because of these options it's an appealing choice to build supercomputer by connecting the computers together.

Once parallel processing computer networks are built next step is to look for suitable parallel programming models. The coding of a suitable parallel program for a given algorithm is strongly influenced by the parallel programming model to be used. Important factors which need to be considered before the usage are programmability, scalability and how well it matches to your computation problem. In order to develop understanding of these models, it is better to look first at the levels of parallelism. And then continue further on.

2.1 Levels of parallelism

2.1.1 Bit level parallelism

In the early stages of processor design speedup is obtained by increasing computer word size. For example, consider that computer word size is of 16 bits and addition of two 32 bits integers needs to done. Only possible way of doing it to first add 16 lower bits with a standard *add* instruction and then add the upper 16 bits using *add-with-carry* instruction. This one extra instruction can be removed if we migrate from 16 bit to 32 bit processor.

2.1.2 Instruction-level parallelism

In general, a computer program is a series of instructions run in sequence by the processors. These instructions can be shuffled and combined in groups which can be executed in parallel. Only precursor condition for instructions shuffling is that there should be no data dependency among instructions. Processor uses concept of pipelining to achieve this kind of parallelism.

Modern processors are divided in to multiple stages – which run in parallel. Each stage corresponds to different action that processor performs according to instruction. Example of a Reduced Instruction Set Computer (RISC) processor is given below. It consists of five different stages Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (IE), Memory Access (MEM) and Write Back (WB). This concept is known as pipelining.

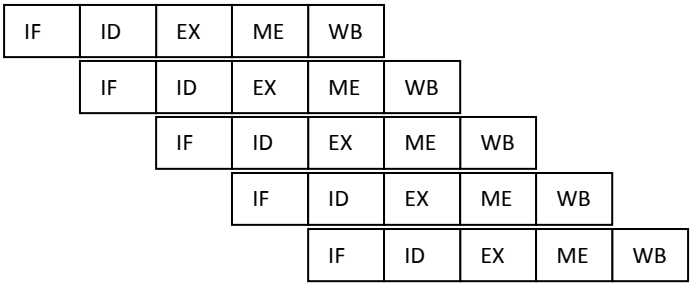


Figure 2.1: Pipeline of RISC processor

2.1.3 Data parallelism

In most data intensive applications, the same operation has to be performed on the large set of data, e.g. add 1 to all elements in the array. This is normally done by iterating through all elements of the array by using a loop. Data parallel programming models exploit this by distributing loop iterations among different threads. Only problem in this approach is data dependencies among different loop iterations. As shown in Figure 2.2 (a) every element of A is dependent upon previous element, i.e. data dependency on preceding element, which makes this loop unsuitable for data parallelism. Whereas in Figure 2.2 (b) no such condition exists which makes it an ideal candidate for data parallelization.

```
for (i =0 ; i< 100 ; i++)
      A [i] = A [i – 1] + 1;
   end for;
```

(a)

```
for (i =0 ; i< 100 ; i++)
      A [i] = A [i] + 1;
   end for;
```

(b)

Figure 2.2: Data dependency example

2.1.4 Task parallelism

In task based parallelism, the code is decomposed into several independent tasks. It's the responsibility of programmer to identify these parallel tasks and convey this information to special run time environment. Run time environment is responsible for scheduling and synchronization of these tasks. Run time environment normally creates one master thread and a number of helper threads, which executes the tasks as they become available. This scheme works fine in the case of independent tasks. For dependent tasks, parent tasks should be finished before proceeding with the child tasks. Two approaches are possible to detect dependency among the tasks. One way requires that programmer should explicitly code dependency of tasks into the algorithm, as for example through locks as in Pthreads or synchronization barriers in OpenMP. And another way is that runtime detects dependency between tasks by looking at task input and output parameters as done in StarSs. [1]

Set of tasks that are ready to be executed are placed inside a queue, from where the idle helper thread selects them. For improving locality and reducing contention among processing cores for accessing queue, a distributed data structure is used. Every thread has its own queue where tasks, ready to be executed, are placed. A thread first looks for tasks in its own queue. If no task is there, it looks for tasks in another thread's queue. If task is found there, it steals it. This mechanism is known as work stealing.[1]

2.2 Programming models

Programming model provides the abstract representation of how data and instructions are stored in memory and how processing of these instructions takes place. In single core processor, most commonly used model is sequential programming model. In this model, there is only one memory in which, both instructions and data are stored. The processor fetches the instruction from memory, decodes it. And access relevant memory location and manipulates data accordingly. In parallel model things are more complex. First of all, we have multiple processors and possibly multiple memory subsystems. Parallel programming model aids programmer mapping applications on parallel architecture. It tries to exploit common features in architectures in order to enable efficient mapping of applications. For portability reasons, they should be independent of specific details of parallel architectures and should remain easy to use. Popular programming models include are message passing, shared memory, data parallel and distributed shared memory.

2.2.1 Message passing model

Message passing programming model as shown in Figure 2.3 (a) is based on the abstraction of parallel computer with distributed address space. Most popular implementation is Message Passing Interface (MPI) [2]. In MPI each processor has access to its local memory only. Other processors can access local memory through explicitly messages passing only. To transfer data from local memory of A to local memory of processor B, processor A should send message containing data to processor B. B should than receive the data into a buffer in its own local memory.

An MPI program is executed by a set of process where each process has its own local memory. Each process gets a unique id, called *rank*. Normally, each process is executed on one processor or core of execution platform. Number of process under execution is fixed at run time. Each process talks to other process using message passing over the network. In most cases, message-passing programming model acts as Single Program Multiple Data (SPMD) i.e. same set of program on multiple chunks of data. But this is not a restriction in the programming model; on the basis of *rank* different process can run different code i.e. Multiple Program Multiple Data (MPMD).

2.2.2 Shared memory model

Many computing platforms such as multi-core platforms offer a shared address space. A suitable programming model for these types of architectures is model in which all threads have access to shared variables. These shared variables can be used for synchronization and data exchange purposes. Figure 2.3 (b) depicts this kind of programming model. Pthreads, OpenMP and SMP superscalar (StarSs) are popular shared memory programming models.

POSIX threads (Pthreads) is a standard for programming with threads based on the programming language C. All the threads of a process have a common address space, which means that all threads can access global and dynamically generated data. Every thread has got its own stack to keep track of functions called and to store local variables. Pthreads are not easy to work with, programmer has to decompose an application to make benefit of it, i.e. rewriting whole sequential program. Race conditions are common occurrence in Pthreads based programs, which makes it a bad choice for productive development.

In contrast to Pthreads, OpenMP provides an incremental way for parallelism i.e. one can change its sequential program to a parallel one step by step. Normally OpenMP is used to parallelize loops. Parallel regions such as loops can be marked with specific compiler directives. When execution enters the parallel region, specific number of slave threads is forked and work is shared among threads (work in case of loops can be number of iterations). After the end of execution region threads are joined again i.e. OpenMP works on fork/join model. As compared to Pthreads, OpenMP is relatively easy to work with. Less lines of code are required to parallelize a program. Sequential code can be parallelized easily without any major rework. On the other hand, race condition can still also occur in OpenMP. In order to prevent race conditions, OpenMP provides a way to mark variables accessed by multiple threads as shared. But it is responsibility of programmer to convey this information.

2.2.3 Data parallel model

Data parallel programming model is shown in Figure 2.3 (c), its name comes from the fact that it processes many data item in parallel in the same way. In this model we have only one executing process which runs the same set of instructions on identical data items. It can be said that this model is the extension of the classical sequential programming model where operations on scalars are replaced by the operations on vectors. Problem with this model is that it doesn't allow independent branching within the process [2]. Thus doesn't allow processing particular data items differently,

which make it unsuitable for certain applications. C* and HPF [6]are example of languages that follows this programming model.

2.2.4 Distributed shared memory model

Distributed shared memory programming model (DSM), also known as the partitioned global address space (PGAS) model can be seen in Figure 2.3 (d). This model tries to achieve the required balance between programmability and exploiting data locality while avoiding the problem of independent branching in the data parallel model. In this model independent thread concept of the shared address space is realized using shared global arrays. These shared global arrays are distributed among the threads. Through specific syntax programmer can dictate array distribution. Access to part of array which is present in the thread memory will be local. One can declare the data to be processed by a given thread in such a manner that it has affinity to that thread. Exploiting locality of access in this manner eliminates or minimizes unnecessary remote accesses from the beginning. Unified Parallel C (UPC) and Co-array Fortran [8] are examples of this programming model.

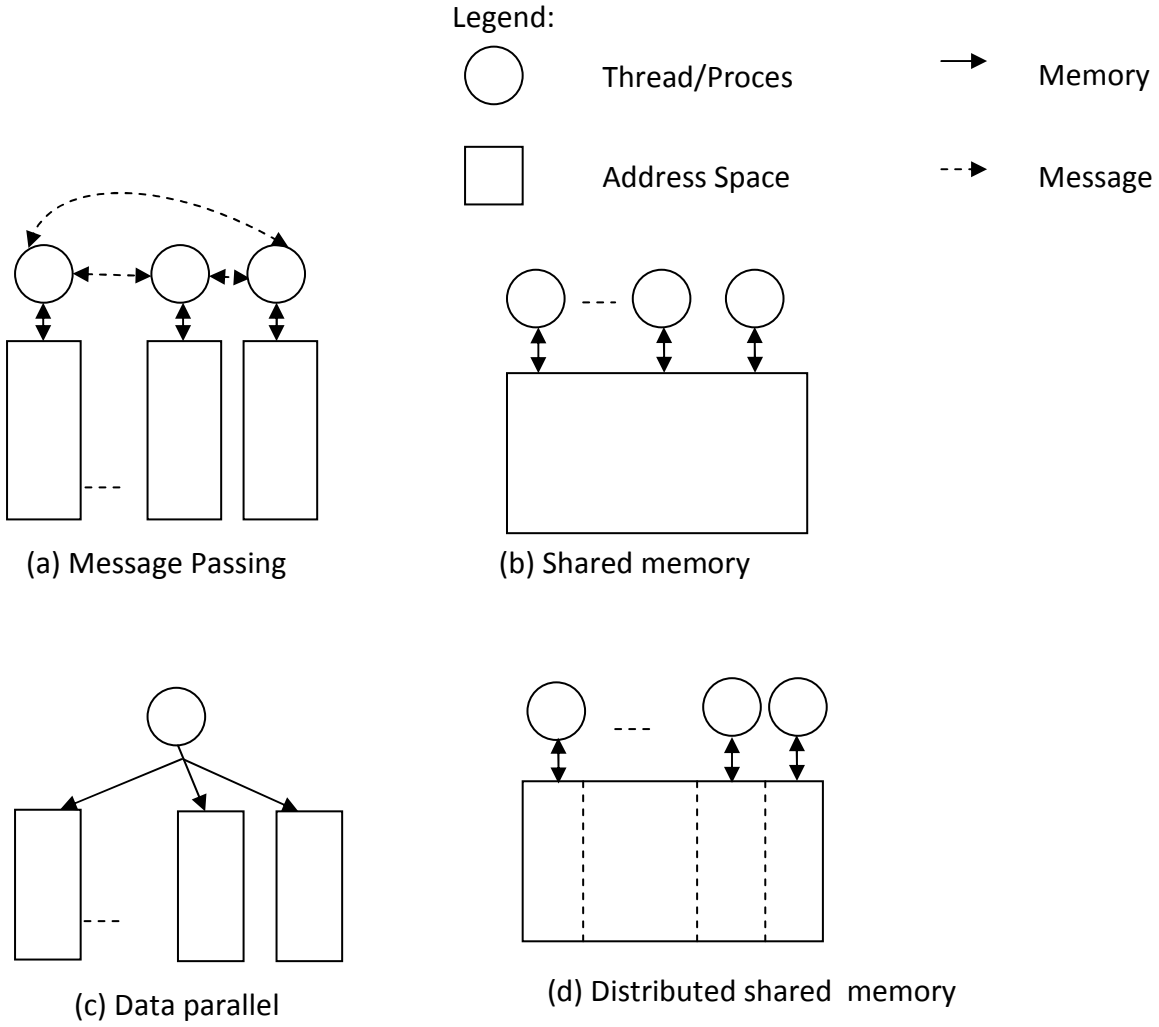


Figure 2.3: Address space and execution in parallel programming models [2]

3 SMP superscalar(SMPSs/StarSs)

In general, a computer program is a series of instruction run in sequence by the processor. These instructions can be shuffled and combined in groups which can be executed in parallel. Only precursor condition for instructions shuffling is that there should be no data dependency among instructions. The StarSs programming model family takes this concept from instruction level to task level. It looks at dependencies among tasks, combined them in groups and executes tasks from different group concurrently on multiple cores.

StarSs provides a programming environment called SMPSs, which was developed specifically for multi-cores and symmetric multiprocessors (SMP) for increasing programmability, portability and flexibility. It improves programmability, as task based parallelization is used and memory locations are easy to reference. Because of shared memory systems, memory locations are accessed, using simple assignment operator.

SMPSs offer a simple programming model, based on the sequential programming which can utilize multiple cores by using automatic parallelization at runtime. The same C code can be compiled by a regular compiler and can run sequentially on the single core machine. Or it can be compiled by the SMPSs compiler, which extracts parallelism, and linked with its run time library to run in parallel on multi-core platforms. Only requirement on the programmer is that application should be composed of coarse-grain functions (called tasks) and these functions should not have any side effects (global variables are not accessed). These functions are identified using annotations in pragmas. Source-to-source translator looks up for these pragmas and generates intermediate C file with some extra information. This information is used by the runtime to parallelize these functions.

SMPSs source-to-source compiler, on the basis of annotated functions with pragmas, separates them from main code. In contrast to other programming models such as OpenMP annotation here does not mean the start of the parallel region. SMPSs run time library builds up a dependency tree on the basis of tasks. Where nodes represents instance of task and edges between nodes specify data dependencies. SMPSs force the programmer to identify directionality of function parameters i.e. input, output and input/output. Dependency graph is built up by looking at function parameters directionality information. Using dependency graph, runtime schedules tasks on different cores. Techniques such as data dependency analysis, data renaming and data locality exploitation are implemented in the runtime to improve performance.

In SMPSs the programmer only specifies functions which can potentially run concurrently. SMPSs will do the data dependency analysis and will determine which functions can run in parallel. In some other programming models such as OpenMP one has to specify these things explicitly. Therefore, SMPSs provides a flexible programming model, which offers an adaptive parallelism influenced by the data dependencies and the cores present.

3.1 Programming model

A SMPSs program is a sequential program where the functions that can potentially run in parallel are annotated with pragmas. In SMPSs nomenclature, these functions are known as the tasks. Annotation with pragmas declares that a function is a task and it also conveys information regarding size and directionality of the parameters. The syntax of the task construct is given below.

```
#pragma css task [clause [clause] ...] function-
declaration
where clause is one of the following:
input(parameter-list)
output (parameter-list)
inout (parameter-list)
high priority
```

- **input clause**: Lists parameters whose input value will be read.
- **inout clause:** Lists parameters that will be read and written.
- **output clause:** Lists parameters that will be written to.
- **high priority clause:** Specifies that the task will be scheduled for execution earlier than tasks without this clause.

Parameters in the directionality clauses (First three clauses) may optionally have dimension specifies with the following syntax:

```
identifier [[expr] [[expr] ...]
```

where identifier is the name of a parameter and expr is a C99 expression. This is required for proper operation of runtime, when the parameter is an array and its size is not present in the parameter declaration. The programming environment consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code with the annotations into standard C99 code with required calls to the supporting runtime library and compiles it using the platform native compiler.

The runtime takes as input the memory address, size and directionality of every parameter at each task's invocation. Further it uses this information to find dependencies between tasks. Whenever in the code a task is called, a node is added in the task graph with a series of edges indicating their dependencies. At the same time it picks up task by looking at the dependency graph and schedules them on the

available cores. The SMPSs runtime uses parameter renaming to remove some data dependencies. This behavior is true for all data types except those of type void *. Which in SMPSs are called as “opaque pointers” as they are not changed by the runtime and are not considered in task dependency analysis.

StarSs runtime detects dependency between tasks by looking at the starting memory address. Consider the case, in which one task access/updates a block of memory and another task which updates/access the middle part of the same block. As StarSs detects dependency by looking at the starting address, which in this case will be different; dependency between the tasks will not be detected. In order to solve this problem, dummy variable can be used to create artificial dependency. These dummy variables are called sentinels in StarSs nomenclature. Sentinels will be placed in the parameter list of both tasks. To creates dependency between the tasks, in one task sentinels will be marked as output where as in another as input.

Once all the tasks have been specified, the next step is how to use them. In order to invoke the tasks, annotated functions must be called within a block surrounded by below mentioned directives.

```
#pragma css start
#pragma css finish
```

These two directives can only be used once in the program. It is not possible to write a start directive after finish directive. These directives are mandatory and all annotated function must be called inside the region surrounded by them.

Race conditions can occur when the data used inside the tasks needs to be manipulated by the master running code outside of any tasks. Dependency tracking by the runtime is not enough to tackle these dependencies. In order to solve this issue, SMPSs provides synchronization directives. One of them is given below

```
#pragma css barrier
```

This synchronization directive forces all tasks generated up till now, should be completed before the master moves further on. In some cases this synchronization can be counterproductive. For example code in (a) has two arrays *a* and *b* which are initialized to 1. Task A and task B performs some operations on these arrays. At the end, inside main code (outside tasks), array *a* is printed out. Task A should be finished, before printing of the array. One way of doing is to use a barrier directive as mentioned above. But if barrier directive is used, it will also wait for task B to finish i.e. inefficient approach. For tackling this problem SMPSs provides below mentioned directive.

```
#pragma css wait on(<list of variables>)
```

In this case main (master) waits until all listed variable values are available. The data unit to be waited on should be consistent with the data unit of the task. For example, if the task is operating on the full range of an array, we cannot wait on a single element arr [i] but on its base address arr. [3]

```
#pragmacss task inout (a)
   void taskA (int a [5])
{
   for (i = 0; i< 5; i++)
   a [i] += 1;
}

#pragmacss task inout (a)
voidtaskB (int b [5])
{
   for (i = 0; i< 5; i++)
      a [i] += 5;
}
void main ()
   {
   int a [5] = {1,1,1,1,1};
         int b [5] = {1,1,1,1,1};

   #pragma start

         taskA (&a [0]);
   taskB (&b [0]);

   #pragmacssbarrier

   for (i = 0; i< 5; i++)
   printf ("%i ", a [i]);

   printf ("\n");
   #pragma finish
}
```

(a)

```
#pragmacss task inout (a)
   void taskA (int a [5])
{
   for (i = 0; i< 5; i++)
   a [i] += 1;
}

#pragmacss task inout (a)
voidtaskB (int b [5])
{
   for (i = 0; i< 5; i++)
      a [i] += 5;
}
void main ()
   {
   int a [5] = {1,1,1,1,1};
         int b [5] = {1,1,1,1,1};

         #pragma start

   taskA (&a [0]);
   taskB (&b [0]);

         #pragma wait on a [0]

   for (i = 0; i< 5; i++)
   printf ("%i ", a [i]);

   printf ("\n");
   #pragma finish
}
```

(b)

Figure 3.1: Wait and barrier directives example code

3.2 Scheduling

Exploiting data locality is one of the major goals in the SMPSs scheduler. For improving data locality, the scheduler makes use of graph information and schedules dependent tasks sequentially to the same core. So that the data present in the cache can be

reused. Scheduler maintains two global ready queues, one is for the high priority tasks and another is for the normal priority tasks. High priority tasks are scheduled as soon as their dependencies are resolved. High priority tasks can be scheduled on any available core. Data locality improvement is not considered while scheduling high priority tasks. Normal priority tasks list is used by worker threads to gather tasks whenever they are idle. Main (Master) thread runs the main code and it creates as many worker threads as cores to keep them busy. Master thread looks up for tasks dependencies and add them to the task graph. If the added task doesn't have any dependencies it is moved to the high priority list, ready to be scheduled by the worker threads. In addition, every worker thread has its own ready list. When a thread finishes running a task, it updates the graph and moves all tasks whose dependencies are resolved to its local ready list. Worker threads priority for fetching tasks, for execution, is given below, where the lowest number represents the highest priority.

1. Look into global high priority queue.
2. Look into own ready queue.
3. Look into global ready queue.
4. Steal tasks form other worker thread queue.

Worker threads while selecting tasks from their own ready queue follow Last In First Out (LIFO) method. They take tasks from global ready queue in First In First Out (FIFO) order. They steal tasks for other worker threads ready queue also in FIFO order. As mentioned in the last paragraph, when worker thread finishes a task - it looks for its child tasks in the dependency graph. Then it updates the dependency graph and brings in all child tasks (form global to local ready queue). For data locality purpose, newly added task whose dependency is just resolved should be selected for execution. LIFO policy for local ready queue serves this purpose well. FIFO policy for global ready queue tries to increase number of tasks, available for execution, by selecting top nodes in the data dependency graph.

As mentioned earlier SMPSs tries to improve data locality. Child nodes in the data dependency graph might be using the data produced by their parent nodes. It is good for data locality purposes that the same core which has executed the parent node (task) also executes the child node (task). Parents and child tasks lies in sequence in the local ready queue and to maintain this structure other tasks steal tasks from other end i.e. FIFO policy. Work-stealing is always done in FIFO order, in order tries to minimize the cache effect. As selected task has spent most time on the queue and has high probability that most of its input data is not present in cache. A snapshot of SMPSs runtime is present in Figure 3.2.

The scheduler design tries to give worker threads different region of the dependency graph to work on, in order to stop accessing the same data for minimizing cache coherency overhead. As long as the worker thread can find ready tasks in the region it is exploring (thread ready queue), if there are unexplored regions in the graph (global ready queue), it will not steal tasks from other worker threads. Thus every worker thread would have independent working set.

In some cases, where communication calls are present inside the tasks, it is better to schedule them as soon as possible. So that more data processing tasks becomes available for execution. If they are present at end of local queue, it is better to change fetching policy for work stealing - so that the idle worker threads could fetch task with communication calls. For tackling this kind of cases, StarSs provides with a runtime switch, to change tasks fetching policy.

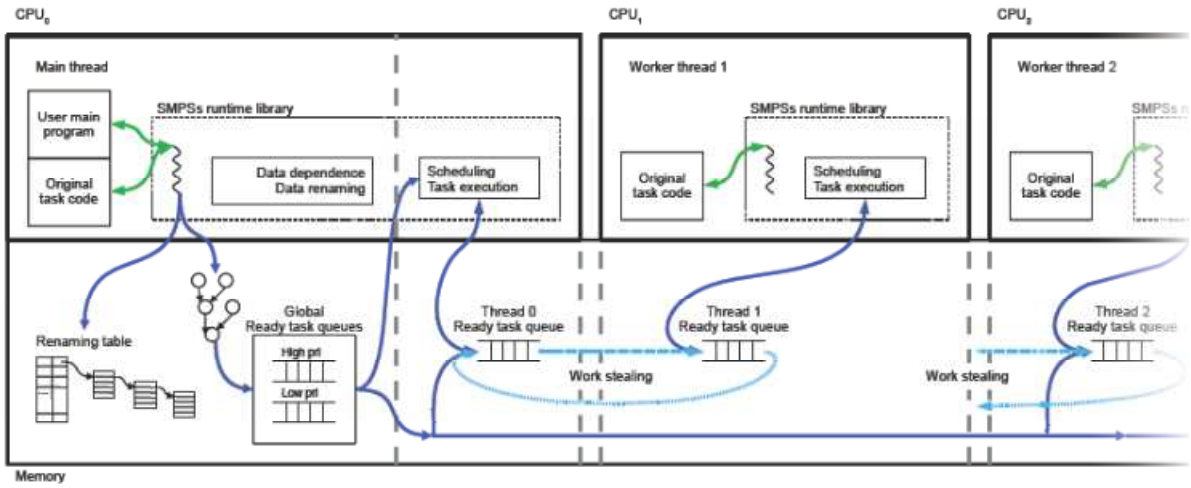


Figure 3.2: SMPSs run time environment [4]

4 Unified Parallel C

UPC is a parallel extension of ISO C, as it inherits most of the features e.g. dynamic memory, pointers etc from ISO C. In addition, it extends ISO C's syntax and semantics for facilitating expression of parallelism. Therefore it's right to say, UPC is a parallel extension of ISO C which uses shared and distributed programming model.[4]

General trend in computing world is to move from uniform shared memory towards distributed memory. But shared programming model has some good features for users. For example, to reading and writing remote memory with assignment statement is more user friendly than using message-passing library. One of objectives while designing Unified Parallel C (UPC) was to make sure that the presence of parallelism and remote access should not make program difficult to understand. Users should be able to see a collection of threads operating in a common global address space and should not worry about the hardware topology. In UPC, a small number of changes to the C language are made, as we have to differentiate between local and remote memory access. Mainly pointers and arrays are the two C constructs which deal with addresses. Introduction of the additional keywords gives the programmer the ability to distinguish between data that is strictly private to a given thread and data that is shared among all threads in the parallel program. In UPC, arrays can be distributed among threads in number of ways, which gives programmer flexibility in the data layout. [5]

4.1 Programming Model

In the UPC programming model, a number of so called threads work independently without any implicit synchronization except that they should start and finish together. These UPC threads may run on different nodes in a distributed memory setting as a cluster. UPC memory and execution model can be seen in figure 4.1. The Integer variable THREADS tells about total number of threads present in the environment. Each thread can get its unique id through integer variable MYTHREAD. THREADS variable is a global constant visible to all threads i.e. same value at each thread. MYTHREAD is a private constant at each thread i.e. different value at each thread. The total number of threads (THREADS) can be specified at either run time or compile time, using appropriate command.

The UPC programming model is a variant of SPMD. Each thread runs the same piece of code. In UPC different threads can run different part of codes, by using conditional

statements based on MYTHREAD identifier. Hence, allowing independent branching for specific set of the data. UPC follows DSM paradigm with some enhancements because it provides private memory for computations on the same node. In UPC, memory consists of the global shared memory space and private memory space. All threads can access any memory location in the global shared memory space. Whereas, private memory space can only be referenced by the local thread only. The global shared space is partitioned among threads, each with an association (affinity) to a given thread. UPC provides programmer, a way to keep shared data affinity with the specific thread that needs it for computation in the future.

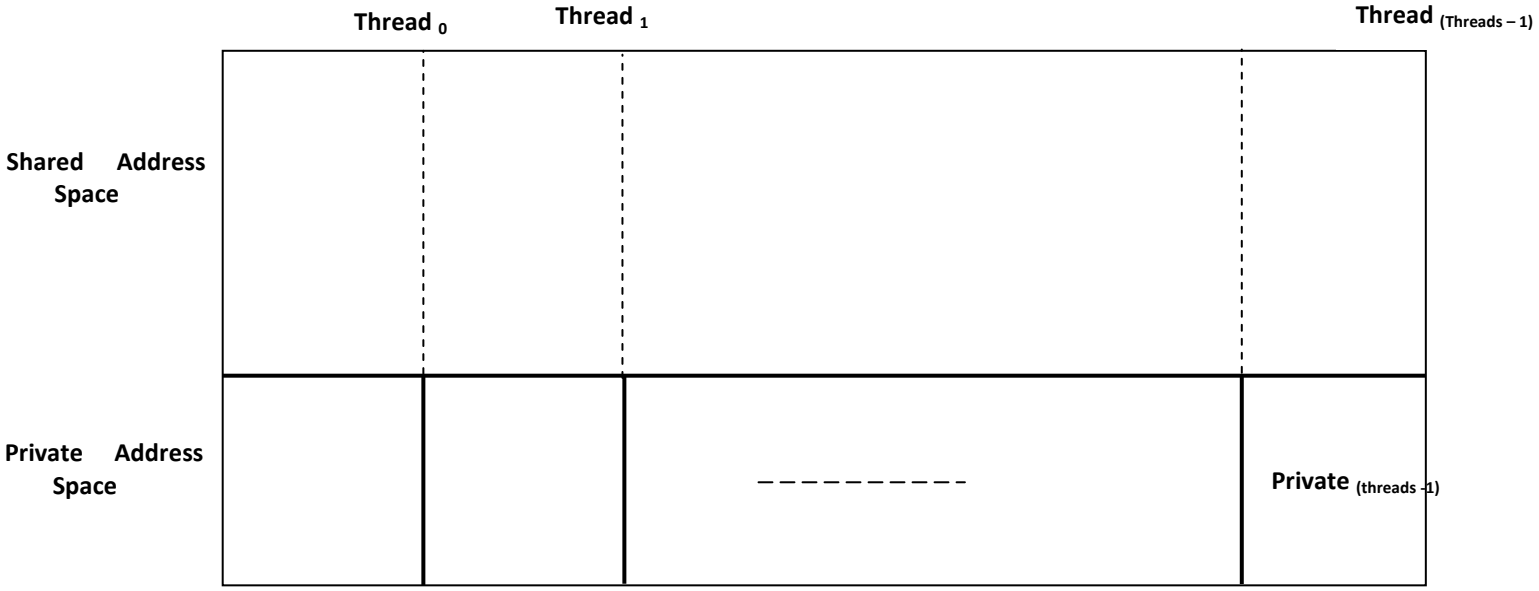


Figure 4.1: UPC memory and programming model [4]

4.2 Shared and Private Variables

This section closely follows the introduction given in *UPC – Distributed shared memory programming book* ([4]).

In UPC, an object/variable could be declared as shared or private. Reserve keyword *shared* is one of the extensions that UPC uses to implement its memory model.

In UPC every thread gets its own copy of private variable; this includes both local and global variables. Thread 0 is unique among all threads because all declared scalar shared objects/variables have affinity to thread 0. UPC treats all standard C style declaration as private variable. For example below mentioned declaration creates a copy of *x* variable for each thread. Each thread can only reference and manipulate its own instance of *x*.

```
int x; //x is private, one x in the private space of each
thread.
```

Reserve keyword *shared* is prefixed, to the standard C declaration for creating shared variables. For example below mentioned example defines a scalar shared variable. This will create only one copy of variable, which can be accessed and manipulated by all the threads. As already stated in previous paragraph, this scalar shared variable will have affinity to thread 0.

For declaring an object to be shared, however, requires explicit use of the sharedqualifier. For example:

```
shared int y; // y is shared, only one y at thread 0 in
the shared space
```

UPC places a restriction on shared variables that they should not have automatic storage. Let suppose shared variable are allowed to have automatic storage. Consider the case in which they are defined inside a function. One thread executes that function updates shared variable contents and exits from the function – variable destroyed. Another thread may access this shared variable, which causes the problem as respective shared variable is already destroyed. To avoid this problem, shared variables are restricted not to have automatic storage.

```
void foo (void)
{
shared int x; // not allowed     ------ (I)
static shared int y; // allowed  ------ (II)
shared int *p; // allowed        ------ (III)
int *shared q; // not allowed          ------ (IV)
...
}
```

Inside above function all declarations which have automatic storage durations are illegal/not allowed. (II) Declaration is allowed as static variables don't have automatic storage duration. (III) Declaration is also allowed as it creates a pointer in private space which points to a memory in the shared space. Statement (IV) creates a shared pointer which points to the private memory. As pointer is created in shared memory space, it is not allowed.

One way of solving this problem is to make statement (I) and (IV) global as shown below.

```
shared int x;
int *shared q;
void foo(void)
{
static shared int y; // allowed
shared int *p;
...
```

```
}
```

UPC allows type conversion between shared and private objects/variables, using cast and assignment it can be done. In general, private objects can't be cast to shared objects and assignment of private to shared objects has undefined results.

4.3 Shared Arrays

This section closely follows the introduction given in UPC – *Distributed shared memory programming book* ([4]).

Shared arrays are placed in the shared global address space. By default, the shared array's elements are distributed among threads in round-robin fashion i.e. first element of array is created in the shared memory that has affinity to thread 0, the second element in the shared space that has affinity to thread 1, and so on. Or in other words, the first element goes to thread 0, the second to thread 1, and so on. The following example declarations demonstrate how a shared vector declaration behaves compared to shared scalar and private scalar declarations.

The declarations

```
shared int x; /*x is a shared scalar and will have
affinity to thread 0 */
shared int y [THREADS]; /*shared array*/
int z; /*private scalar*/
```

For four threads, default layout is shown in Figure 4.2, where x and y were placed in the shared space and z copies were placed in the private memory space of each thread. If the statement

```
shared int y [THREADS];
```

was replaced with

```
int y [THREADS];
```

Then every thread will have its own complete private version of the array y.

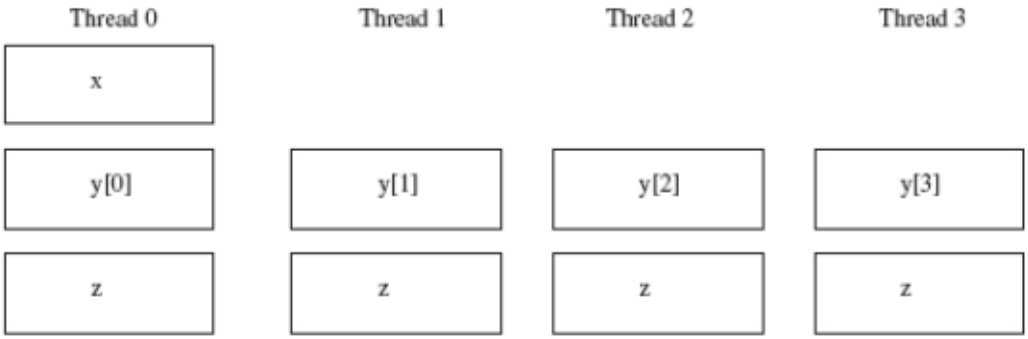


Figure 4.2: Memory layout of variables[4]

In the case of higher-dimensional arrays, the elements of a shared array are still distributed in round-robin way. For example, the statement present below will result into layout shown in Figure 4.3.

```
shared int v [4][THREADS];
```

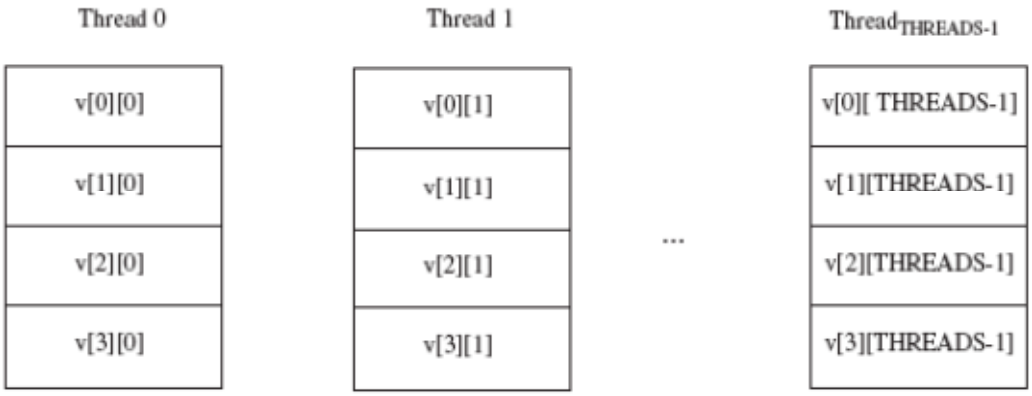


Figure 4.3: Array distribution among threads for block size 4 [4]

The default shared array distribution scheme may not be appropriate for optimal execution, in some cases. A different approach for distributing array elements could improve data locality exploitation and execution efficiency. Shared array default distribution can be changed by mentioning a given block size, also known as blocking factor. Declaration for altering the default distribution is given below.

```
shared [block-size]array [number-of-elements]
```

For example:

```
shared [4] int a [16];
```

In above statement, array a [] has 16 elements which are distributed among four threads. First four elements of array will go to thread 0; next 4 elements will go to thread 1 and so on. Block size and total number of threads (THREADS) determines affinity to threads using following equation.

$$\left\lfloor \frac{i}{blocksize} \right\rfloor \textbf{mod } threads$$

```
shared [3] int x [12];
```

Above statement has a blocking size of 3, which means that array elements in a block of 3 are distributed across the threads in round-robin way. The resulting layout for 3 threads is shown in Figure 4.4.

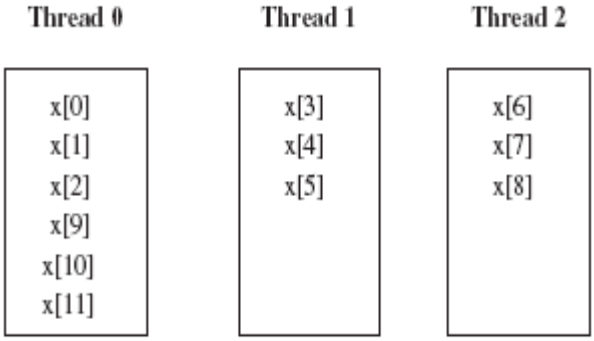


Figure 4.4: Array distribution among threads for block size 3[4]

If the previous statement is changed to

```
shared [12] int x [12];
```

Then all array elements would have affinity to thread 0. Omitting the block size or making it zero in the brackets would result in making all array elements have affinity to thread0. Using such indefinite block size, the previous statement's result/effect can also becreated.

```
shared [] int x [12];
```

or

```
shared [0] int x [12];
```

In many cases it is desirable that array's data should be distributed in contiguous blocks such that, whenever possible each thread should get one of the chunks. One way of doing of it is to put * in block size place. For example,

```
shared [*] int y [8];
```

would produce the layout shown in Figure 4.5 for the case of three threads. This works in the same way with two- and higher-dimensional arrays as in the case of one-dimensional arrays.

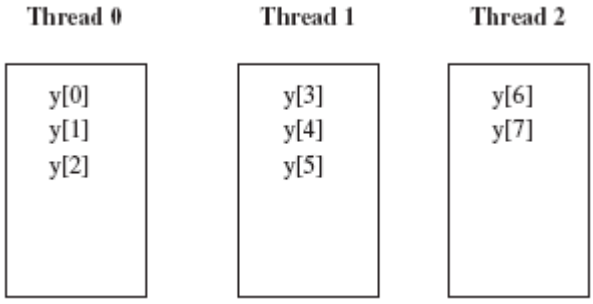


Figure 4.5: Contiguous array distribution among threads [4]

```
shared [3] int A [4][4];
```

In the case of above statement, array elements are blocked by a factor of 3. Therefore, blocks of three elements each is distributed across the threads in round-robin fashion untilall the array elements are allocated. The resulting layout in the case of four threads is shown in Figure 4.6.

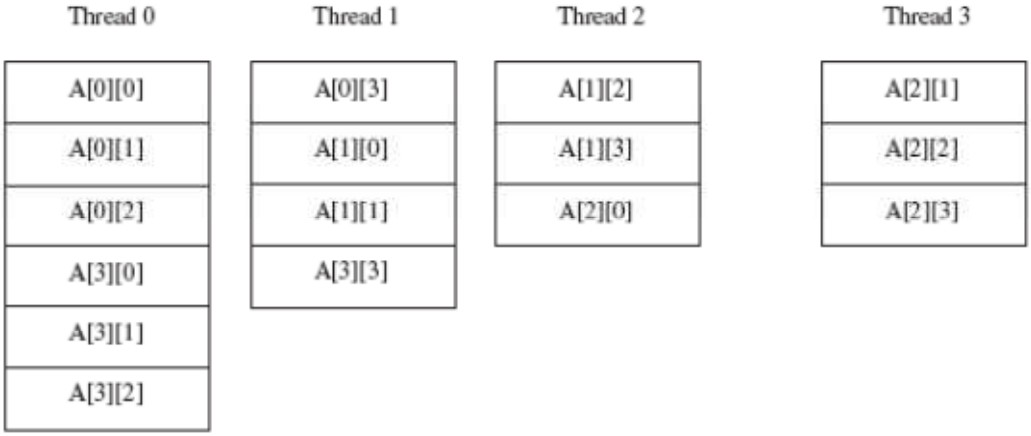


Figure 4.6: Two dimensional array distribution among threads[4]

4.4 UPC Pointers

This section closely follows the introduction given in *UPC – Distributed shared memory programming book* ([4]).

UPC has similar syntax for pointer declarations as in ISO C. But because of memory model which is partitioned between shared and private memory space. UPC pointers can be divided into four major classes as shown in Figure.

Where does the pointer reside?	Private	Shared
Private	PP	PS
Shared	SP	SS

Legend :

PP – private to private
PS – private to shared
SP – shared to private
SS – shared to shared

Figure 4.7: Pointer classes[4]

```
int *p1; // private to private
```

Above declaration declares a pointer, which resides in private memory space and can be used to point memory location in private space.

```
shared int *p2; // private to shared
```

Above declaration declares a pointer, which resides in private memory space and can be used to point memory location in shared space. As it lies in private memory space, every thread will get one private copy of this pointer.

```
int *shared p3; // shared to private
```

Here a pointer which lies in shared memory space and points to a memory location in private memory is declared. It is against the principle that shared space should be visible to all threads and private space should only be visible to respective thread; therefore, it should be avoided.

```
shared int *shared p4; // shared to shared
```

In above statement p4 is a shared pointer pointing to the shared space; thus, it has one instance with affinity to thread 0.

Memory region where the pointers mentioned above are located and to which region they are pointing to is shown in Figure 4.8. There exists one copy of each pointer in each threads private space. Only one instance of P3 and P4 is created in the shared space with affinity to thread 0. Each of the p1 pointers points to its associated private space and can also point to the shared space that has affinity to that pointer. Each of the p2 pointers can point anywhere in the shared data space. The pointer p4 can also point anywhere in the shared space. As a shared pointer, p3 has only one instance created in the shared space of thread 0.

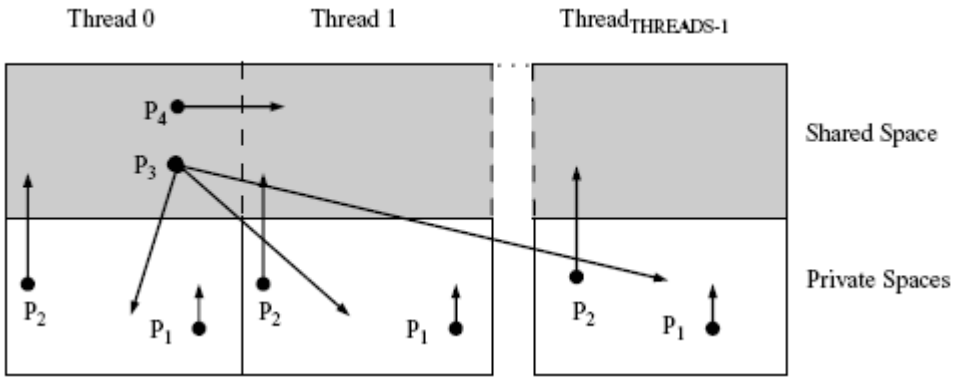


Figure 4.8: Pointer classes memory layout and referencing space [4]

5 Implementation details and results

For evaluating performance of hybrid UPC and StarSs version, I first start with implementing a matrix multiplication with StarSs. I try out row and square block data distributions. Then after that I start looking up for optimal block size and data distribution required to get best performance. After that I start developing matrix multiplication code using UPC for distributed memory and StarSs for shared memory i.e. hybrid code.

For testing purposes, I have used NEC Nehalem cluster installed at HLRS. This cluster has 700 computing nodes connected together using a InfiniBand network. Each node has two sockets with Intel Xeon Processor X5560 Quad core running at 2.8 GHz with 8 MB. Software used and their respective versions can be seen in Table 5-1.

Softwares	Version
gnu	4.6.1
Open MPI	1.5.3
OpenMP	3
Berkeley UPC translator	2.13.6
Berkeley UPC run time	2.13.6
SMPSs/StarSs	2.4

Table 5-1: Software and their versions

Berkeley UPC implementation 2.13.6 (translator/run time) has limitation of block size 1048576 (2^{20}). So, some values in the graphs (where this limit is crossed) in the next sections will be missing.

5.1 StarSs

Matrix multiplication algorithms can be divided into two major stages. For example consider the case in which two matrices A and B are multiplied and resultant matrix is stored in C. First phase is the initialization phase in which A and B is initialized with random values. Second one is the computation phase where C matrix is calculated. In both phases, the same set of operations or same piece of code will be executed on different sets of data. Hence, these two phases can be written as two functions.

Definition of these two functions along with StarSs pragmas notation is mentioned below. These functions accept address of memory locations to identify the set of data to work on. StarSs run time uses memory address location and parameter directionality info to buildup dependency graph.

```
#pragma css task input (n) output(subMat)
void initWithRand (double *subMat, int n)
n: Number of elements to be initialized.
subMat: Starting address of respective Matrix sub-block.
#pragma css task input(subMatA , subMatB) output(subMatC)
void multiply (double *subMatA, double *subMatB, double
*subMatC)
subMatA: Starting address of respective Matrix A sub-
block.
subMatB: Starting address of respective Matrix B sub-
block.
subMatC: Starting address of respective Matrix C sub-
block.
```

Once the tasks inside the matrix algorithm are identified, the next step is to look for appropriate data distribution among tasks. I first tried row block distribution for simplicity reason and to reduce number of cache misses. Data distribution of matrices in this case is present in Figure 5.1. As shown in Figure 5.1 with shaded blocks, for calculating results of one row block of matrix C, respective row block of matrix A and complete matrix B is required. This means that initialization of respective matrix A's block and complete matrix B should be finished before calculation can be started. StarSs builds up the dependency graph (shown in Figure 5.2) by looking at the starting address of data set. In order to calculate sub-block C1 multiply task needs complete matrix B and sub-block A1. But multiply task receives starting address of sub-block A1 and sub-block B2, so dependencies created by the sub-block A1 and sub-block B1 could only be identified. But multiply task needs complete matrix B - other sub-blocks (B1, B2, B3) of matrix B might not be done with initialization. To avoid this problem, a StarSs barrier pragma is used between initialization and computation phase.

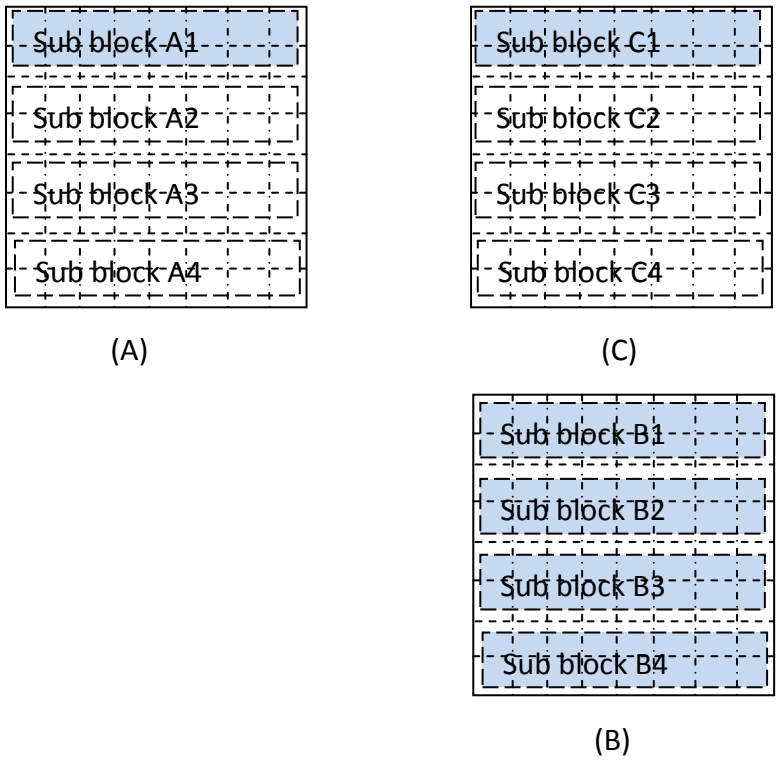


Figure 5.1: Row block distribution of matrices

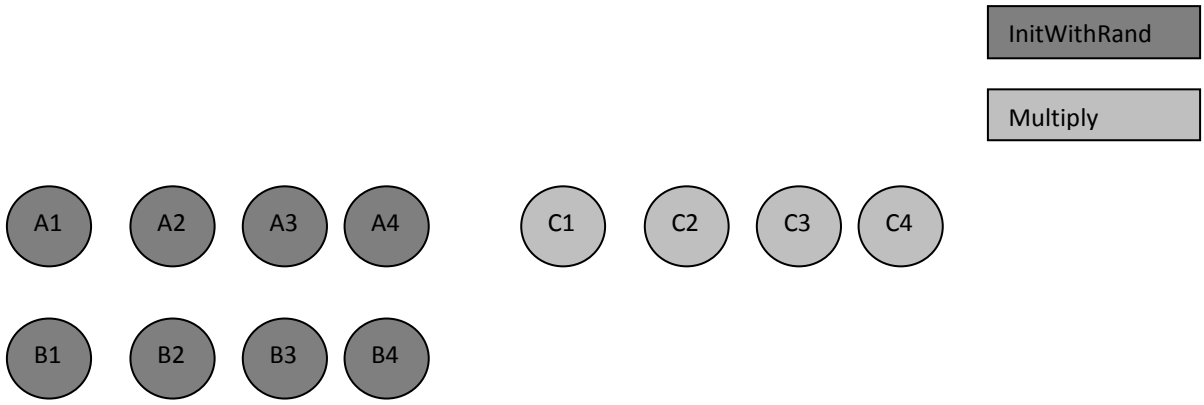


Figure 5.2: Dependency tree for row block distribution

Another distribution I tried is block-wise as shown in Figure 5.3. Barrier statement in row-wise distribution introduces synchronization control which can cause performance degradation. To calculate results for one block of matrix C, all respective blocks of matrix A and B need to be multiplied and added to C as shown in Figure 5.3. In this scenario, we have two initialization and one computation dependency (previous computation of C should be finished before next can start). Here computation of sub-block C1 requires, only sub-block A1 and B1. And multiply task is

provided with the starting address of sub-block A1 and B1. So, no StarSs barrier pragma is needed; all dependencies can be detected by looking at the starting address. Dependency map is presented in Figure 5.4. In contrast to the row-wise distribution, here multiple tasks calculate same block of matrix C. Thus matrix C needs to be initialized first, so a new function mentioned below is added.

```
#pragma css task input (n) output(subMat)
void initWith0 (double *subMat, int n)
n: Number of elements to be initialized.
subMat: Starting address of respective matrix C sub-block.
```

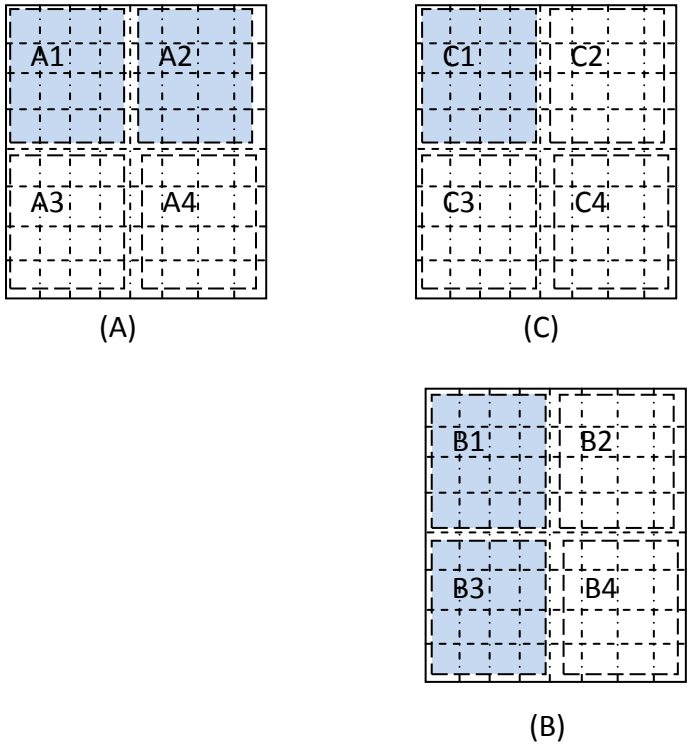


Figure 5.3: Block wise data distribution of matrices

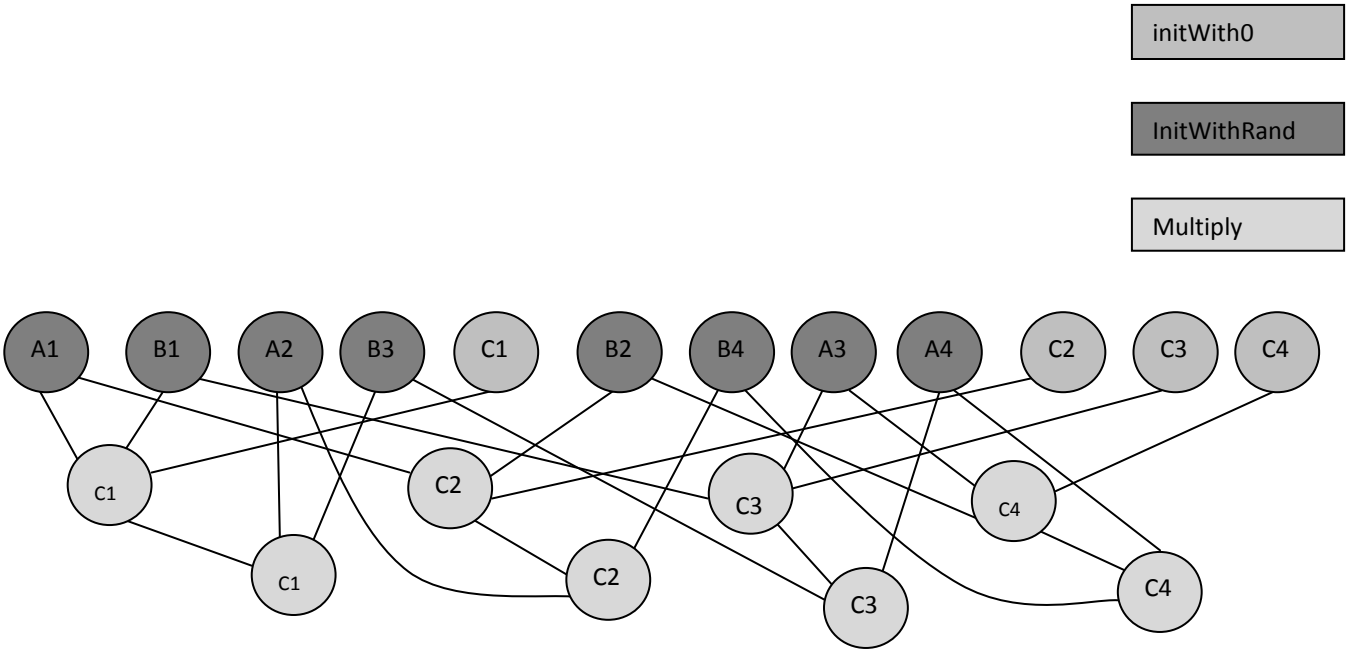
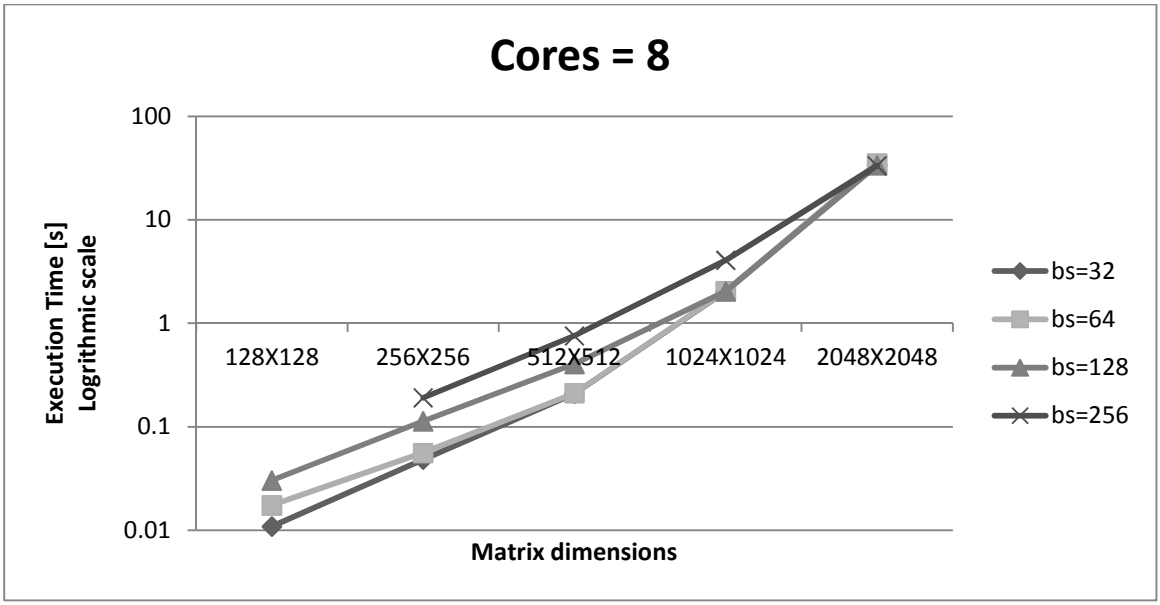
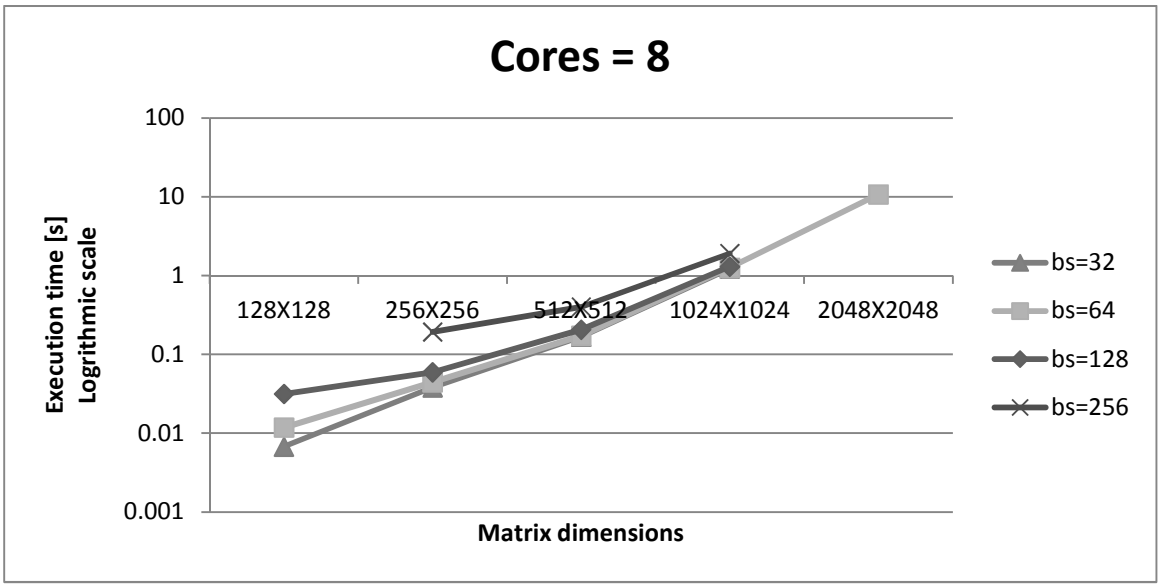


Figure 5.4: Dependency tree for block wise data distribution

For evaluating the performance, I execute above mentioned approaches with multiple matrix dimensions and block sizes, which can be seen in Figure 5.1. One of the factors influencing performance is the size of the blocks as this is the which is number of rows assigned to each task for working on in the case of row wise distribution and dimensions of block in the case of square block distribution. It can be seen from Figure 5.1 that when dimensions of block are increased, execution time also increases. Because number of tasks decreases with increase in block dimensions, so not enough work is available for workers. Same case is true for block wise distribution in Figure 5.2. By looking at the Figure 5.1 and Graph 5.2, it can be judge that we get good results for block size of 32 and 64. As difference between them is not so significant, I took block size of 64 for further analysis.



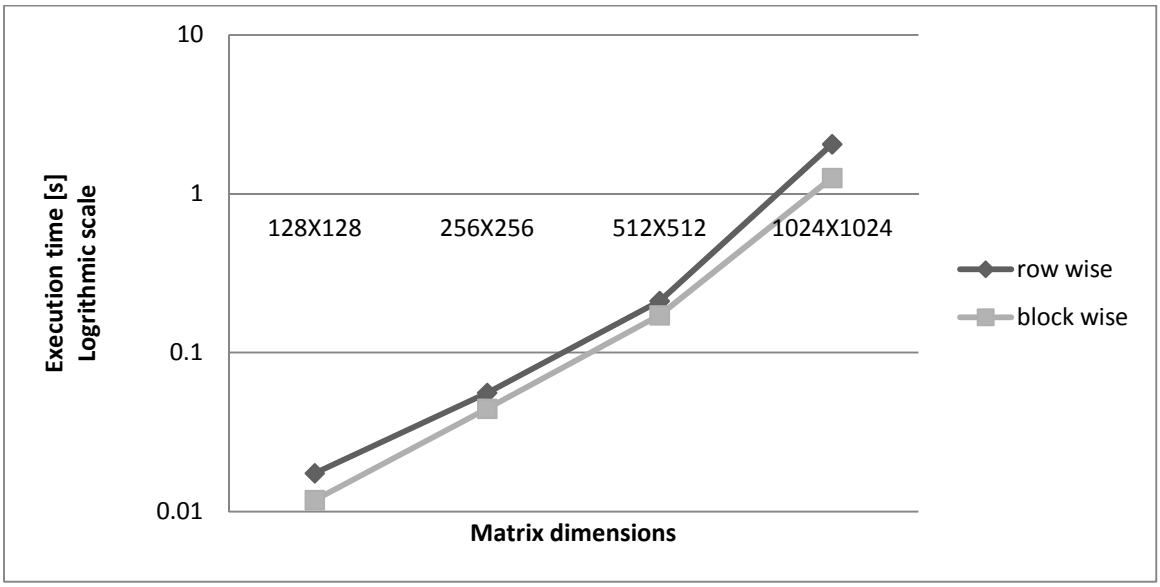
Graph 5.1: Row block distribution



Graph 5.2: Square block distribution

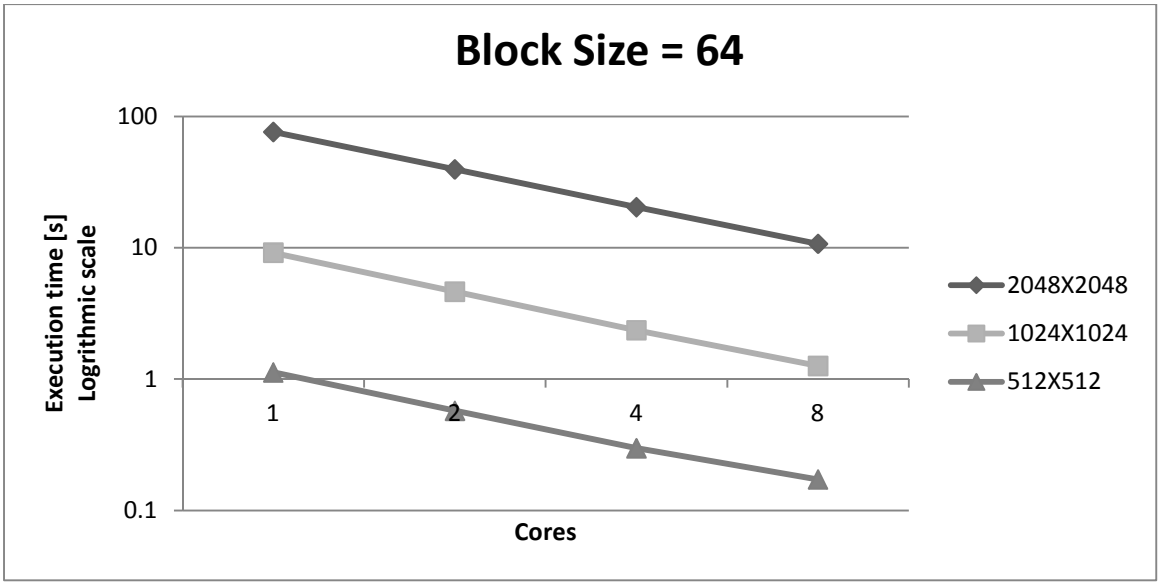
Graph 5.3 tries to compare execution time of matrices for row wise and block wise data distribution. Only block size of 64 is taken as it shows good results, as mentioned above. It can be seen clearly that block wise distribution performs better than row wise. One of the reasons, for better performance of block wise distribution is that there is no barrier statement. The computation tasks (`multiply`) don't have to wait

for all initialization tasks (initWithRand) and (initWith0) to finish. They can start execution as soon as their respective blocks are ready.

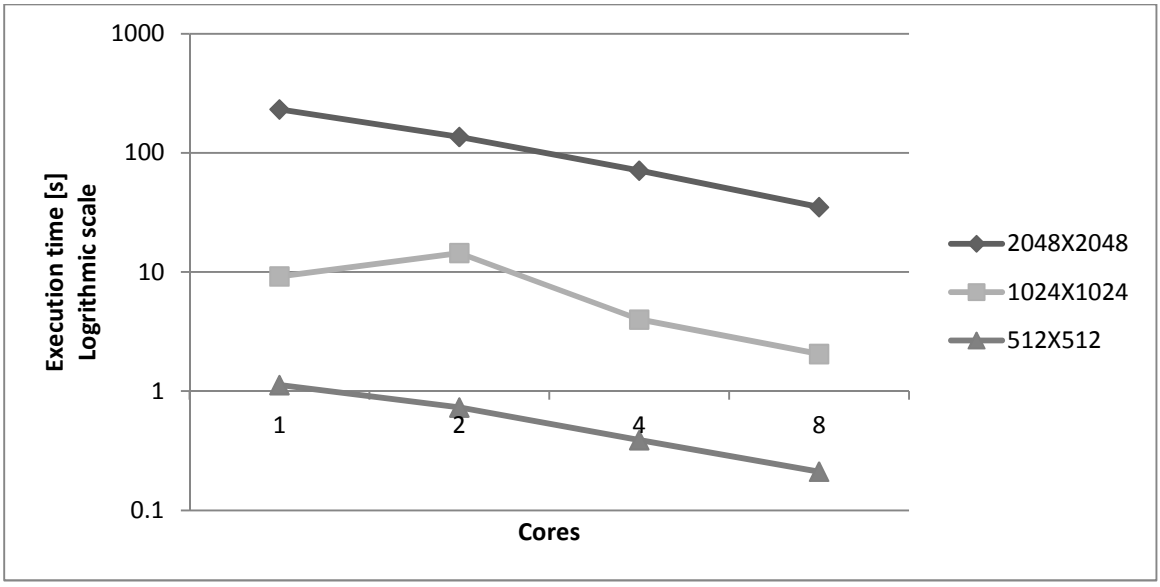


Graph 5.3: Row wise and block wise data distribution

Graph 5.4 and Graph 5.5 looks at the scalability of StarSs; number of cores is increased from 1 to 8, for row wise and block wise distribution. From the graphs they look like perfectly scalable, if enough work is available for all the workers. For block wise distribution, in the of matrix size 256X256 not enough work is available for all workers. So it doesn't scale any further when core number is increased from 4 to 8.



Graph 5.4: Scalability plot (square block distribution)



Graph 5.5: Scalability plot (row block distribution)

5.2 UPC

For evaluating UPC, the SUMMA matrix multiplication algorithm is implemented using UPC specific constructs.

Pseudo algorithm of SUMMA

Every UPC thread initializes its part of A and B block.

For (Traverse over all the required blocks – shaded ones in Graph 5.2)

If (both respective A and B blocks ready)

Compute respective C block.

Else if (any A or B block not available/ready)

Wait for respective A or B block.

Compute C block.

End if

End for

Figure 5.5 shows the matrix storage scheme in the memory. First all the sub-blocks of matrix are stacked up as shown in Figure 5.5 (b). Then each sub-block is stored as a one-dimensional array as shown in Figure 5.5 (c). Hence two levels of indexing need to be done to reach the desired element location.

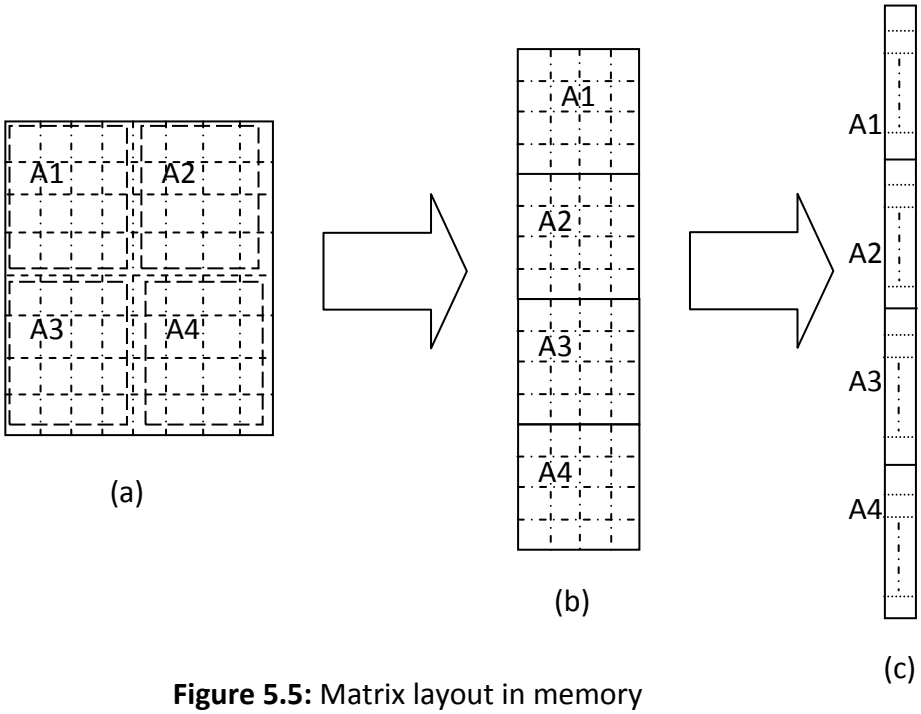
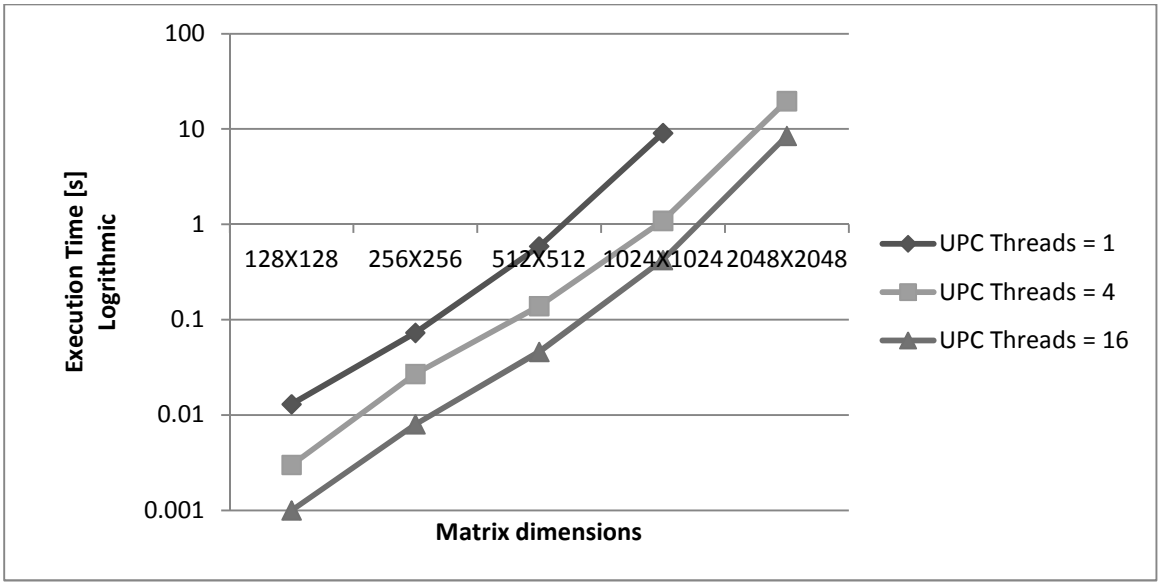
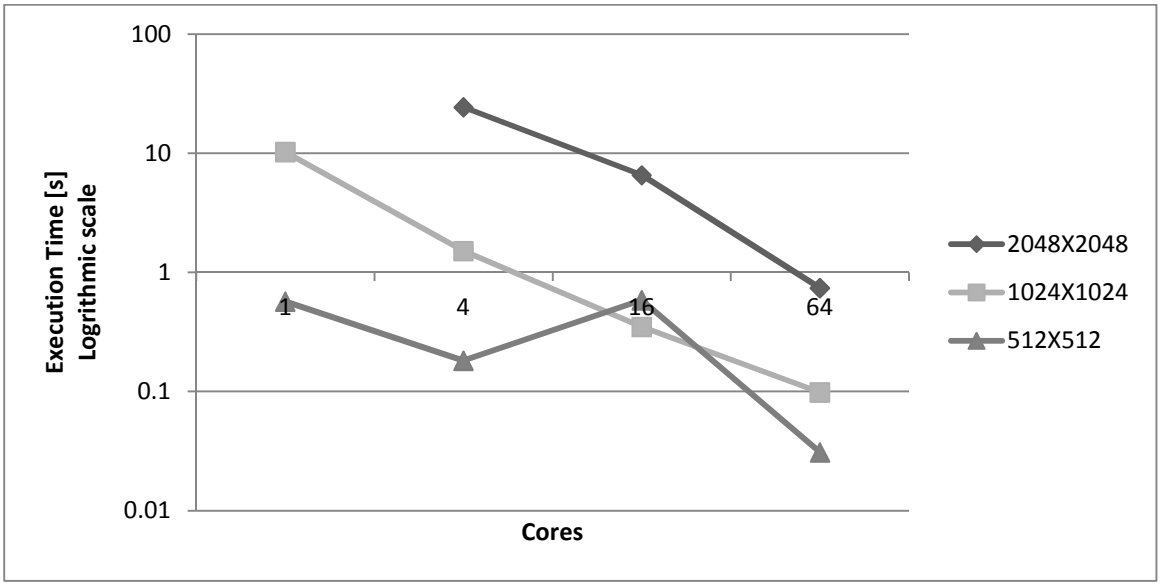


Figure 5.5: Matrix layout in memory

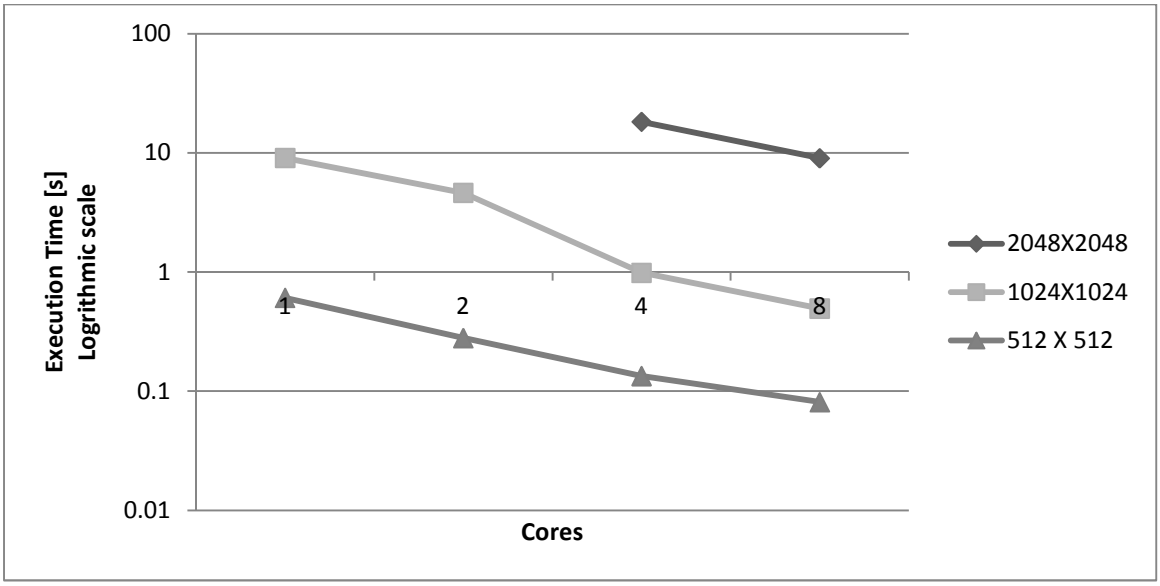
First, square block wise distribution shown in Figure 5.3, for matrix multiplication is used because it gives good results in the case of StarSs. In order to check scalability, different matrix dimensions for a range of threads are tried out, its execution time is noted down and results are plotted in Graph 5.6. Graph 5.7 shows UPC threads across multiple nodes. Execution time decreases if we move from 1 node (8 cores) to 2 nodes (16 cores). Because of the Square block distribution we can only have threads in multiples of 4 i.e. 1, 4, 16, …, 4^n. Computer nodes used for testing purposes has got only 8 cores. It means that when number of threads is increased from 4 to 16 - 16 threads will be running on 8 cores which can lead to the performance degradation. To avoid problem of threads only in the powers of 4, row wise data distribution shown in Figure 5.1 is implemented. Results of this implementation are shown in Graph 5.8. Here we have only 8 threads for 8 cores, so performance degradation (because of the core over-subscription) is eliminated. As mentioned in the start of section 5 Berkeley UPC implementation used has limit on block size. So, some points, in Graph 5.7 and Graph 5.8, for matrix dimensions 2048X2048 are missing.



Graph 5.6: UPC matrix multiplication (using block wise distribution) on one node



Graph 5.7: Scalability plot(using block wise distribution) on multiple nodes



Graph 5.8: Scalability plot (using row wise distribution) on one node

5.3 StarSs + UPC

The SUMMA matrix multiplication algorithm is implemented to check performance of hybrid (StarSs + UPC) version. Here two levels of data distribution are done, one for distributed memory (on UPC threads level) and second is for shared memory (on StarSs tasks level). Block-wise data distribution is performed on both levels which can be seen in Figure 5.6. Previous results show that StarSs performs well for block dimensions 64 X 64. So on StarSs tasks level block dimensions are set to 64 X 64. Whereas on UPC threads level block dimensions of block depends on the number of nodes.

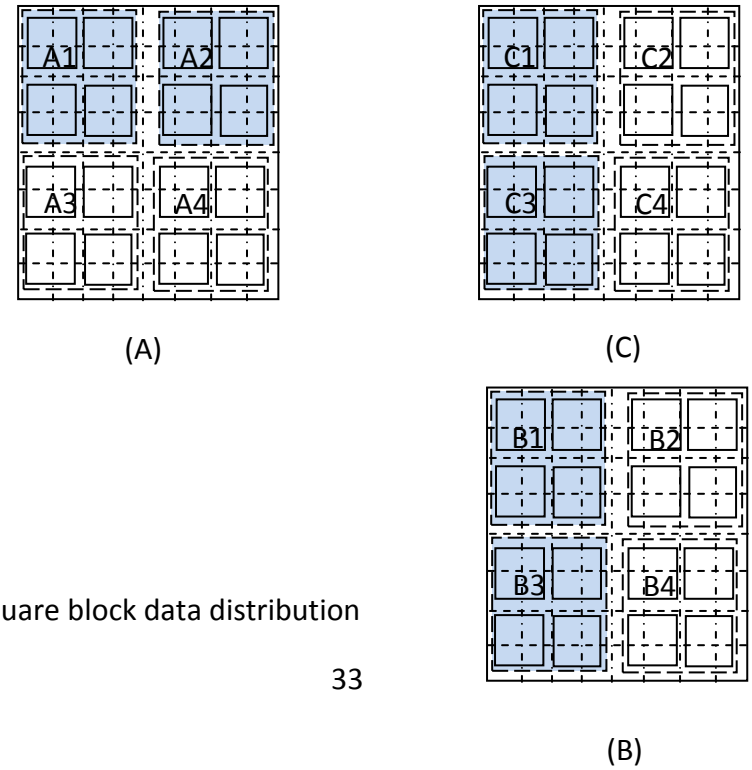


Figure 5.6: Matrix square block data distribution

Pseudo algorithm for hybrid SUMMA

Call initRand for all Matrix A StarSs-level blocks.
Call initRand for all Matrix B StarSs-level blocks.
Call initZero for all StarSs-level C blocks.
Call checkAllBlksInit for matrix A.
Call checkAllBlksInit for matrix B.

For (Traverse over all the required blocks – shaded ones in

Graph 5.2)

Call copyRmtBlkA.
Call copyRmtBlkA.
Calculate all 2nd level C blocks. //StarSs tasks

End for

To store matrices the same storage scheme as mentioned in Figure 5.5 is used. The following global arrays are used to store matrices.

```
shared double [blockSize] A [rows * cols];
shared double [blockSize] B [rows * cols];
shared double [blockSize] C [rows * cols];
blockSize: number of elements in a block i.e. 16 elements
for matrix in Figure 5.5.
```

As already discussed in section 5.1, the first phase in matrices multiplication algorithm is the initialization of the matrix. In hybrid approach UPC threads, running on different nodes, will access each other's UPC-level data block. A way needs to be established to inform adjacent UPC threads when initialization of a local block is complete. A new shared array is declared to indicate status of UPC-level blocks.

```
shared int [1] 1stLvlBlks [THREADS];
```

In hybrid version, initialization is done by a number of StarSs tasks. Above array (1stLvlBlks) element can only be set if all the tasks are finished. So a local array is needed to keep track status of all StarSs-level data blocks for both A and B matrix.

```
int 2ndLvlBlksA [y];
2ndLvlBlksB [y];
```

y: Number of the StarSs-level blocks i.e. 4 for matrix in Figure 5.5.

Initialization task (`initRand`) defined in section 5.1 is modified to accommodate above new requirements i.e. respective location of 2ndLvlBlk.

```
#pragma css task output(ptr, blkCmpl);
void initRand (double *ptr, int *blkCmpl);
```

```
*blkCmpl: Holds address of respective 2ndLvlBlk's
location.
```

Initialization task (initZero) for initialization of matrix 0 remains same.

```
#pragma css task output(ptr)
void initZero (double *ptr);
```

New StarSs task is added to check when all sub-blocks are initialized and set corresponding location in 1stLvlBlks global array in order to notify other threads.

```
#pragma css task input(blkCmpl, mat)
void checkAllBlksInit (int*blkCmpl, char mat);
int *blkCmpl: Will hold starting address of array
2ndLvlBlks.
char mat:  Required matrix name needs to be checked.


#pragma css task input(blkLoc, locA) output(strLoc, AFlg)
void copyRmtBlkA (double *strLoc, intblkLoc, intlocA, int
*AFlg);
double *strLoc: Temporary location for storage.
int blkLoc: Which block from shared memory needs to be
copied.
int locA: Which 1stLvlBlks [threads] needs to be checked.
int *AFlg: Used to introduce dependency.
```

New StarSs tasks are needed to fetch required blocks from UPC threads, running on remote nodes, and store them in a temporary location for later used by the multiply task (mul). These routines copy UPC-level block from remote nodes. In order to stop multiply task (mul) for continuing further on before the results are copied, an artificial dependency is created by using AFlg, BFlg variables.

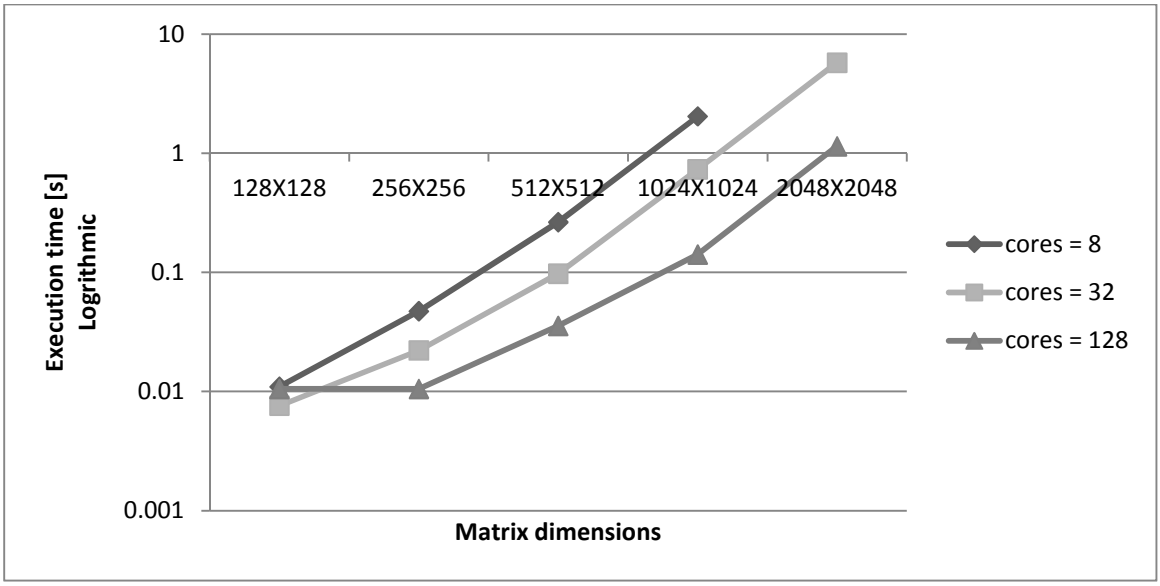
```
#pragma css task input(blkLoc, locB) output(strLoc, BFlg)
void copyRmtBlkB (double *strLoc, intblkLoc, intlocB, int
*BFlg);
double *strLoc: Temporary location for storage.
int blkLoc: Which block from shared memory needs to be
copied.
int locB: Which 1stLvlBlks [threads] needs to be checked.
int *AFlg: Used to introduce dependency.
```

Multiply task (mul) almost remains the same with some additional variables to create artificial dependency.

```
#pragma css task input(A, B, AFlg, BFlg) inout(C)
void mul (double *A, double *B, double *C,int *AFlg, int
*BFlg);
double *A: Holds starting address of matrix A StarSs-Level
block.
```

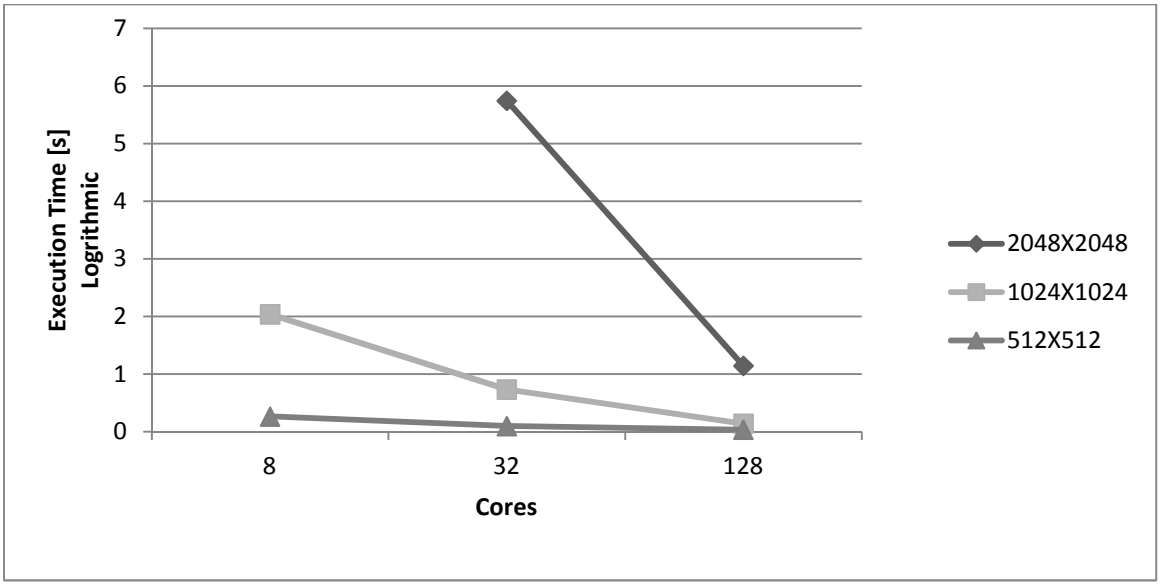
```
double *B: Holds starting address of matrix B StarSs-Level
block.

double *C:Holds Starting address of matrix C StarSs-Level
block.

int *AFlg: Used to introduce dependency.

int *BFlg: Used to introduce dependency.
```

Graph 5.9 shows improvement in execution time when number of cores is increased. In the case of 128 cores, there is no improvement when we move from 128X128 to 256X256. As matrix 1st level block size is smaller and more time is spent in moving the blocks (communication) rather than computing the results i.e. over head of communication becomes prominent in the case of small data transfers. Same argument can also be made for not having linear graphs for 32 and 128 cores. As mentioned in the start of section 5 Berkeley UPC implementation used has limit on block size. So, a point in the case of matrix dimensions 2048X2048 in Graph 5.9 and Graph 5.10 is missing.



Graph 5.9: Improvement in performance by increasing number of cores - square block-wise distribution.

Graph 5.10 looks at the scalability of UPC for various matrix sizes. Communication network is more optimized for sending large data transfers. So less speedup can be seen for matrix size: 1024X1024 etc, when we move from 32 to 128 cores, as compare to matrix size: 2048X2048.



Graph 5.10: Scalability plot (multiple nodes) - square block-wise distribution.

5.4 MPI

Use of virtual topologies simplifies the program structure and makes code understandable, provided that data distribution matches the virtual topology. MPI provides us with multiple virtual topologies such as graph and Cartesian. Data distribution in Figure 5.3 resembles the Cartesian topology so it is used. Figure 5.7 shows the block distribution among different process, i.e. process with Cartesian coordinates (0, 0) holds first block of all matrices.

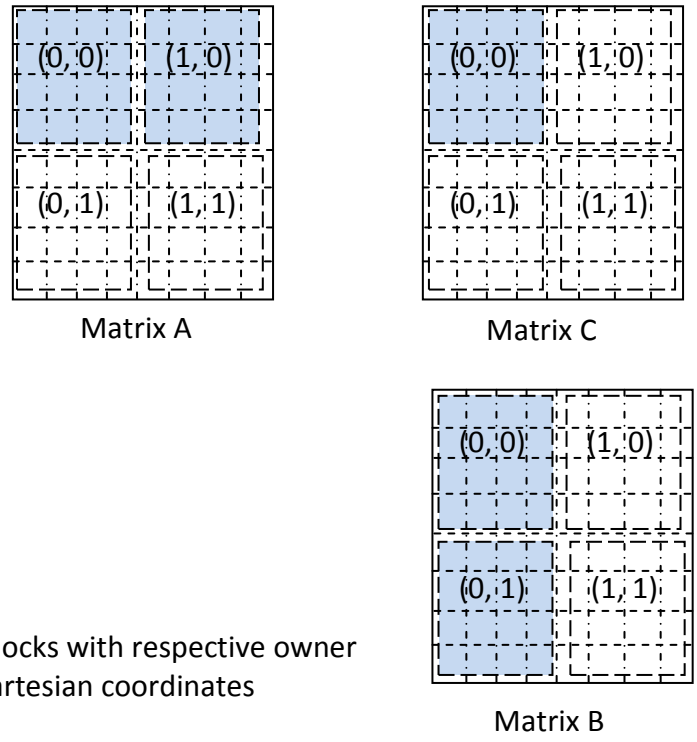


Figure 5.7: Data blocks with respective owner process Cartesian coordinates

In the case of matrix multiplication, every process needs to share its data block with all other processes in its respective row and column. For example, process with Cartesian coordinates (0, 0) will share its data block with all blocks in its row group (Figure 5.8 (b)) and with (Figure 5.8 (c)). Or it can be said that every process broadcasts its data block among its respective row and column groups. So, all rows and column process are grouped together for ease of use.

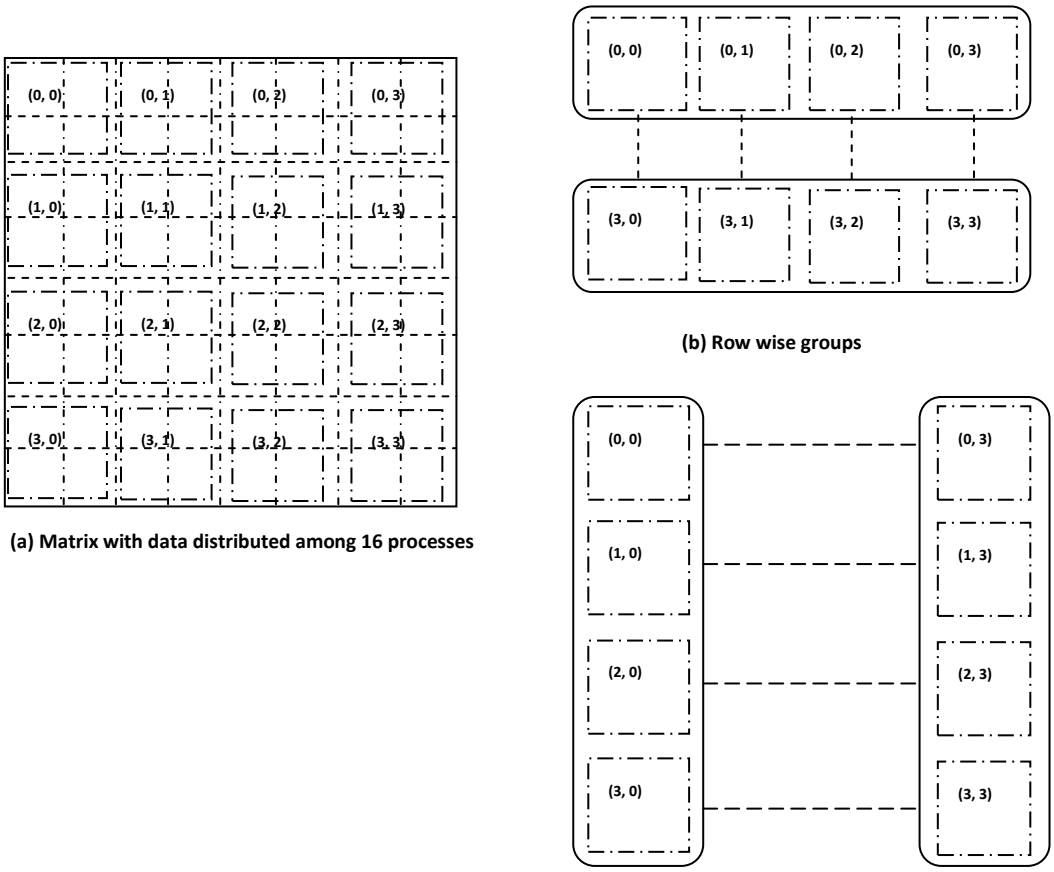


Figure 5.8: Row and column grouping of data blocks

Next step is determining the communication strategy among row and column process. MPI provides a number of options i.e. blocking send/receive, non-blocking send/receive and collective communication calls. Non-blocking send/receive and collective communication calls are singled out for further usage. Non-blocking send/receive is given preference as they allows an asynchronous program flow i.e. if data is available, computations can be done alongside waiting for the next block of data. Collective communication calls, use fine/optimal communication algorithm which gives good result – normally difficult to achieve using send/receive calls.

In non-blocking send/receive communication algorithm every process send its data block to its right neighbor process (in case of row group) or its downward process (in case of column group) and receives data block from left process (in case of row group) or up process (in case of column group). For example, the process with Cartesian coordinates (0, 1) sends it block to all other process in its row and column. And

receives blocks from all other process in its rows and columns. This sending and receiving pattern can be seen in Figure 5.9.

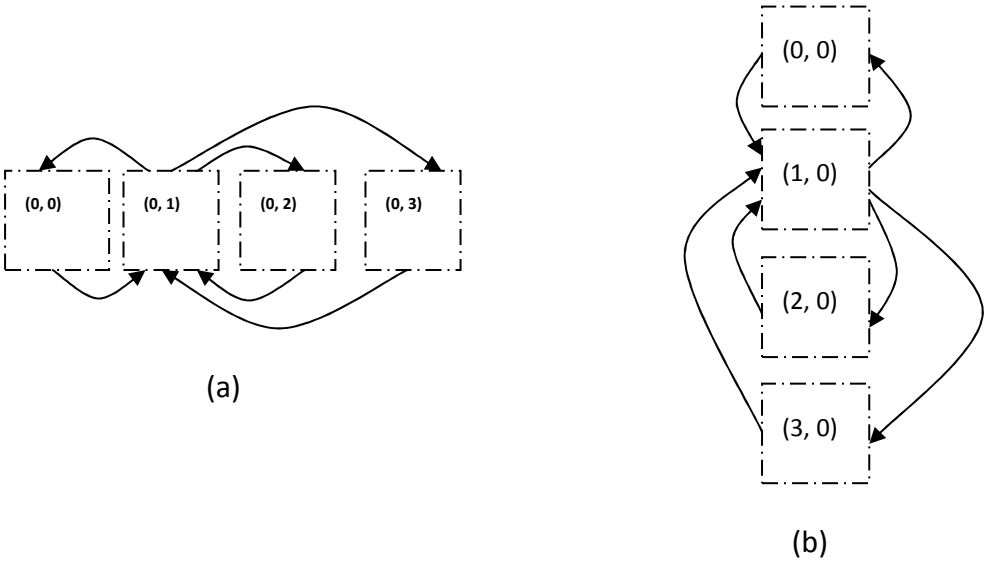


Figure 5.9: Non blocking send/recv messages for process with Cartessian coordinates (0, 1)

Pseudo Algorithm for MPI approach using non-blocking send/receive

Call MPI_Cart_create (vu, …) // creating Cartesian topology.
Create row and columns groups as mentioned Figure 5.8.
Every process set receives for all row and column blocks.
Every process initializes its assigned block.
Every process sends its block to all row and column process.

For (Traverse over all the required blocks – shaded ones in Figure 5.7)
As soon as required blocks available compute respective C block.

End for

In the second communication scheme, all non-blocking send/receive is replaced by the collective communication call *MPI_Allgather.* As already known from discussions in previous sections, for the calculating a block of matrix C, all respective row blocks of matrix A and column blocks of matrix B are needed. So collective communication call *MPI_Allgather* is used to collect respective row or column blocks.

Pseudo Algorithm for MPI approach using MPI_Allgather

Call MPI_Cart_create (vu, …) // creating Cartesian topology.
Create row and columns groups as mentioned in Figure 5.8.
Every process initializes its assigned block.
Everyprocess gets required data using all-gather-all.
Everyprocess computes its data block as shown in Figure 5.7.

For (Traverse over all the required blocks – shaded ones in Figure 5.7)
Every process computes its respective C block.

End for

Final result matrix (C) is distributed among all the process as shown in Figure 5.10 (a). As processes are arranged in Cartesian topology, we can take benefit of it for gathering final matrix. First all root process (first process) in each row collects matrix using *MPI_Gather* call, so all row blocks of matrix are present at first column group Figure 5.10 (b).After that first process in the first column group gathers complete matrix using *MPI_Gather* call.

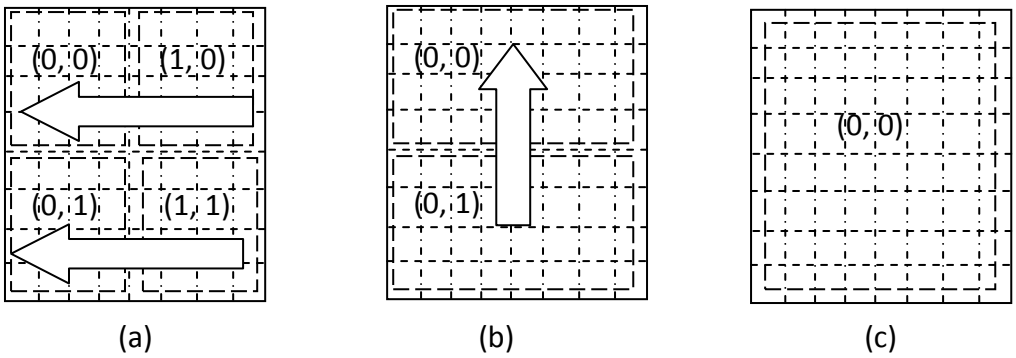
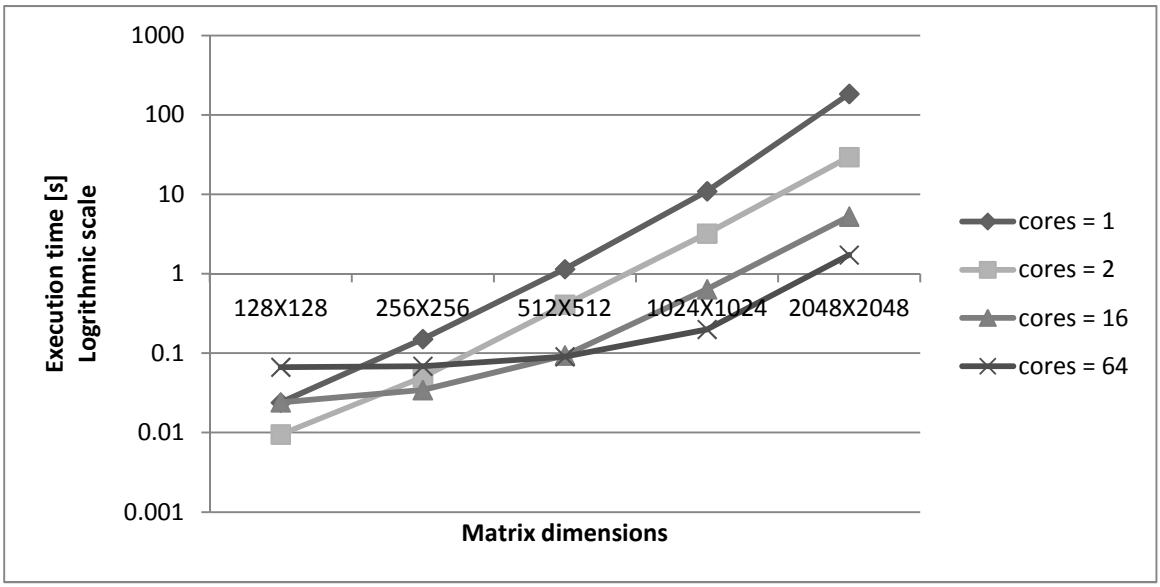
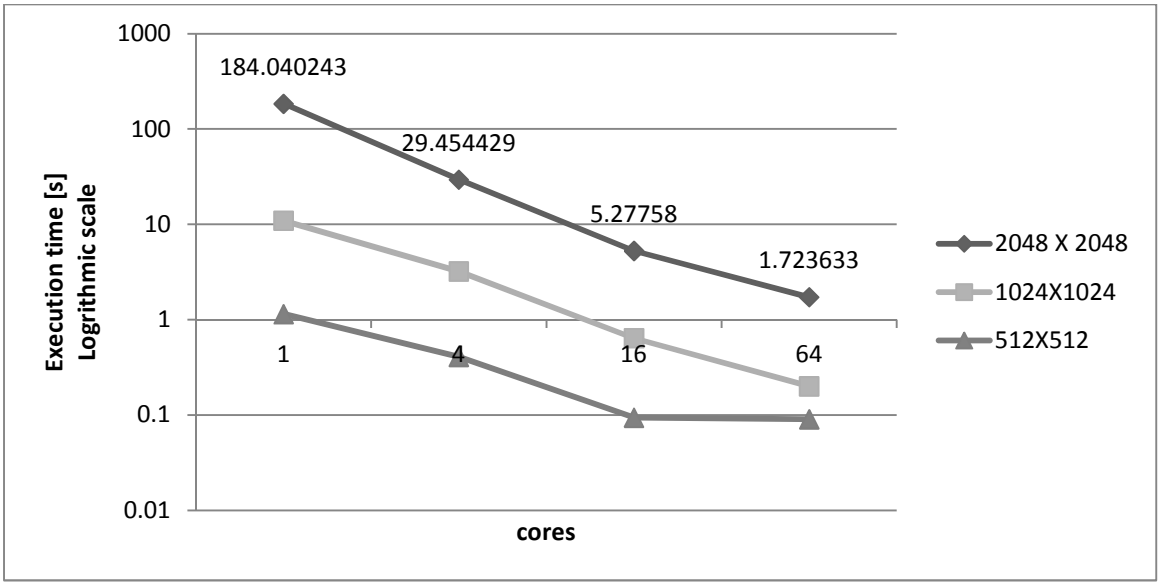


Figure 5.10: Result matrix (C) gathering scheme.

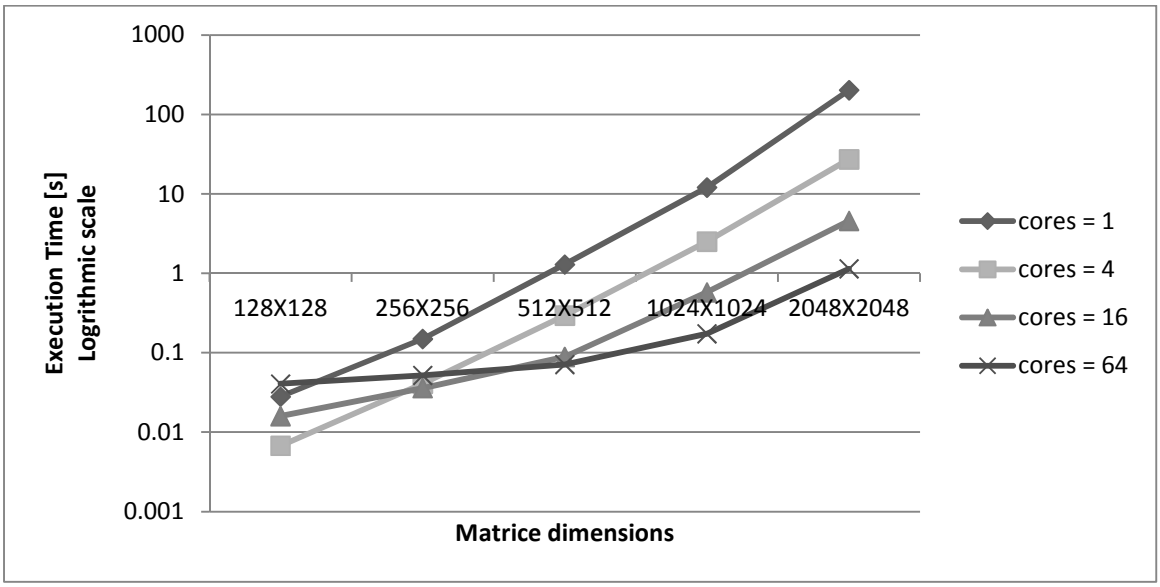
Graphs with execution time for various matrices dimension are given below. It can be seen from Graph 5.15 that for intra node communication *MPI_Allgather* performs better as compare to non-blocking approach.



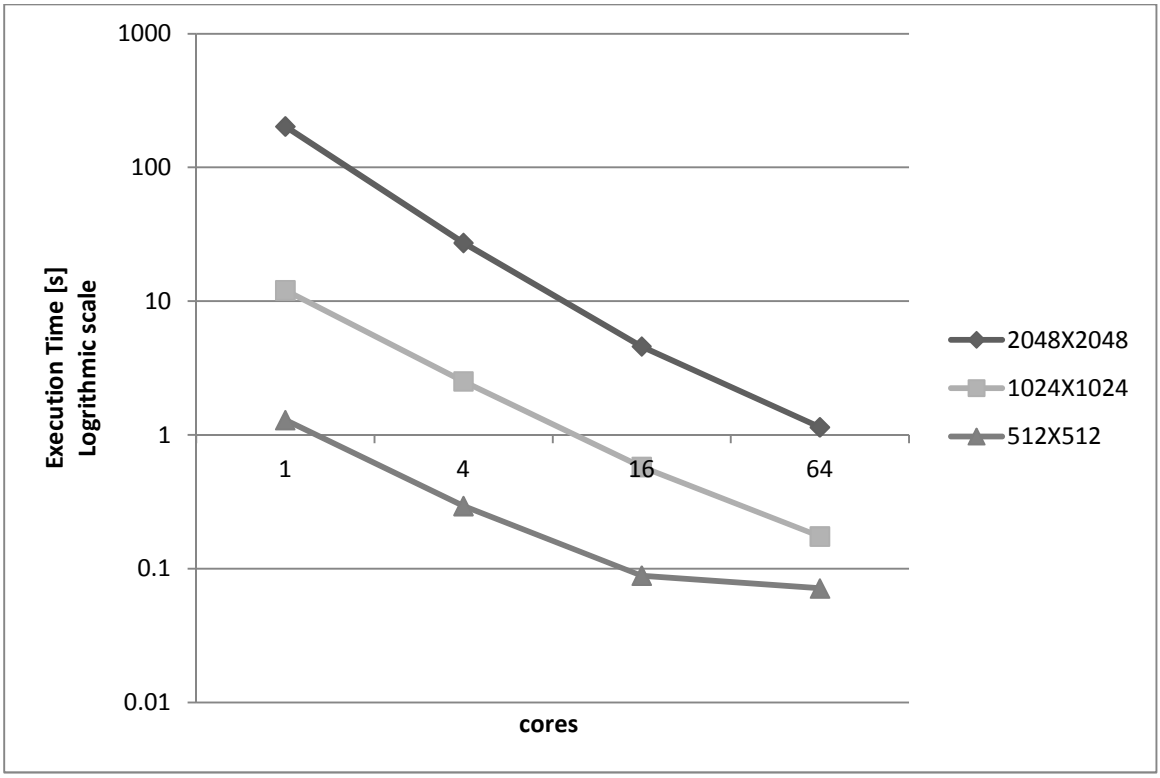
Graph 5.11: Execution time for various matrix sizes (non-blocking send/recv)



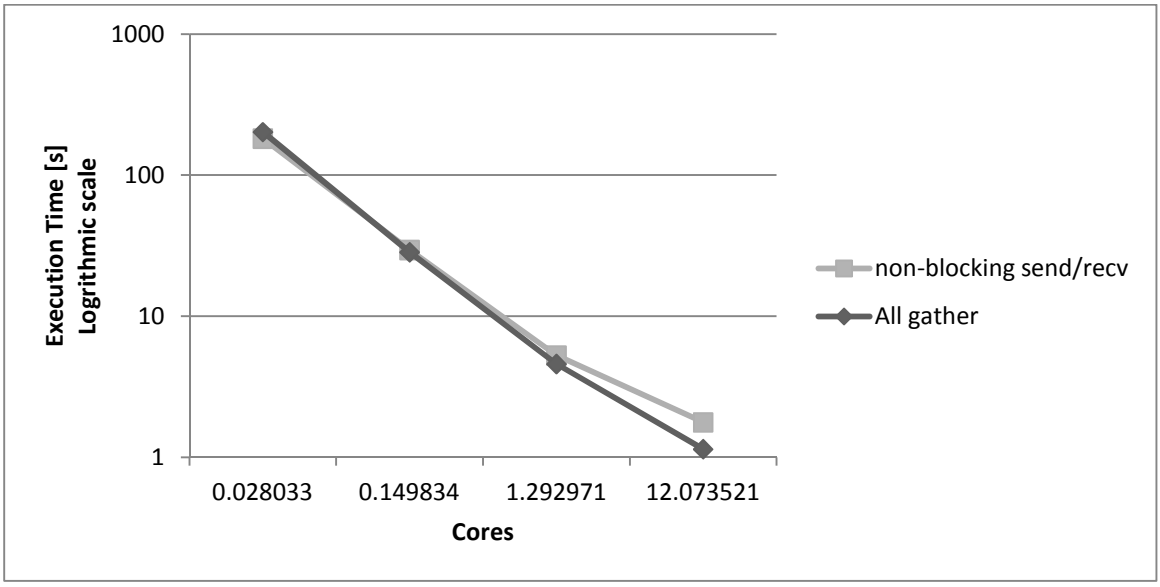
Graph 5.12: Scalability plot (non-blocking send/recv)



Graph 5.13: Execution time for various matrix sizes (all gather)



Graph 5.14: Scalability plot (all gather)



Graph 5.15: Comparison for matrix size 2048X2048

5.5 OpenMP

OpenMP provides us with the incremental way to parallelize sequential programs. Sequential matrix multiplication algorithm, implemented using loops, is parallelized using OpenMP compiler pragmas. Iteration of the loops (initialization and

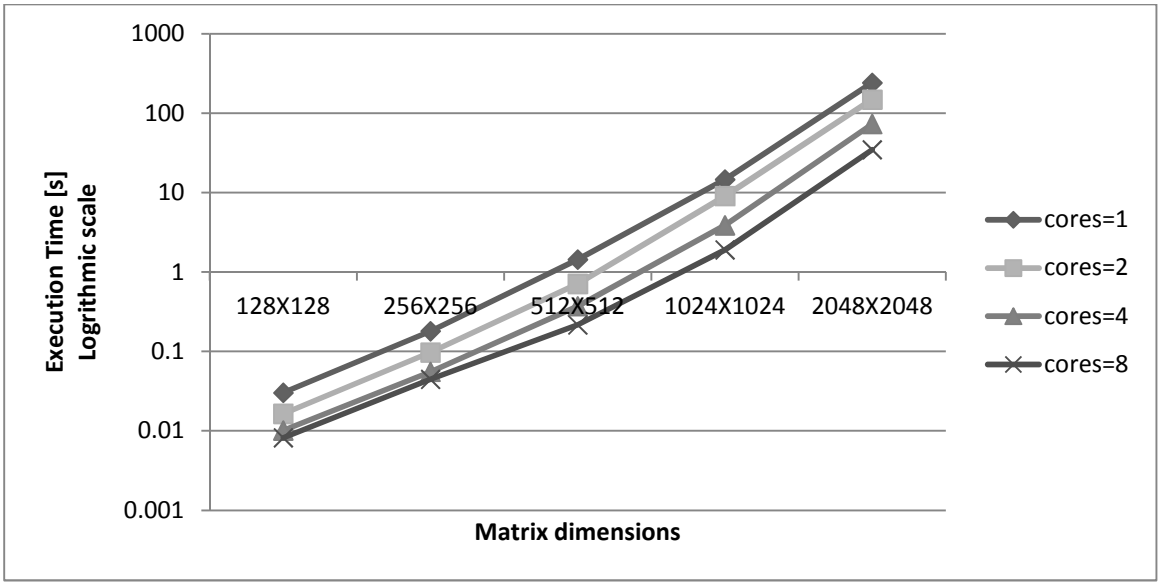
multiplication) is equally divided among OpenMP threads running on different cores. For initialization only one loop is needed, so the matrices array is divided among OpenMP threads to work on. For matrix multiplication, three loops are present, only outer loop is parallelized. Iteration of the outermost loop will be divided equally among all OpenMP threads; or it can also be said that each thread will calculate specific number of C matrix rows.

Pseudo code for matrix multiplication using OpenMP

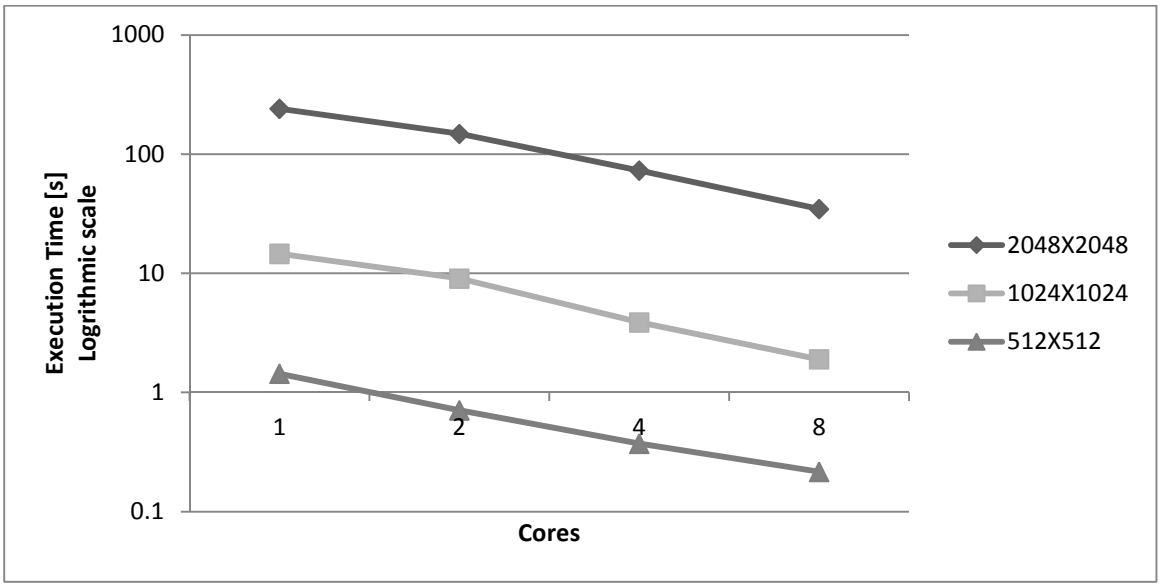
NRA: Number of rows of A.
NCA: Number of columns of A.
NCB: Number of columns of B.
CPUS: Number of CPUS i.e. 8 in our case.

```
#pragmaompfor schedule (static, NRA / CPUS)
forі=0; i< NRA; i++
forj=0; j< NCA; j++
                a [i] [j] = rand ();
        end for
end for

#pragmaomp for schedule (static, NRB / CPUS)
forі = 0; i< NCA; i++
for j =0; j< NCB; j++
                b [i] [j]= rand ();
        end for
end for

#pragmaompfor schedule (static, NRA / CPUS)
forі = 0; i< NRA; i++
forj=0; j< NCB; j++
                fork=0; k< NCA; k++
                        c[i][j] += a[i][k] * b[k][j];
                end for
        end for
end for
```

Performance of OpenMP can be seen in Graph 5.16 and Graph 5.17. In contrast to the above mentioned approaches, i.e. StarSs in section 5.1 and UPC in section 5.2, OpenMP perform well even for small matrix sizes. Reduction in execution time can be observed for all matrix sizes, when core number is increased.



Graph 5.16: Matrix execution time for various cores



Graph 5.17: Scalability plot

5.6 MPI + OpenMP

From section 5.4 it is learnt that collective communication calls perform better. So it is used for developing hybrid version i.e. MPI for distributed memory systems and OpenMP for shared memory systems. In the MPI code, initialization and matrix computation loops are parallelized using OpenMP directives as described in the previous section. MPI is used for parallelization at the node level (at the distributed memory level) and OpenMP is used for parallelization at the cores level (at the shared memory level).

Pseudo Algorithm for MPI + OpenMP approach

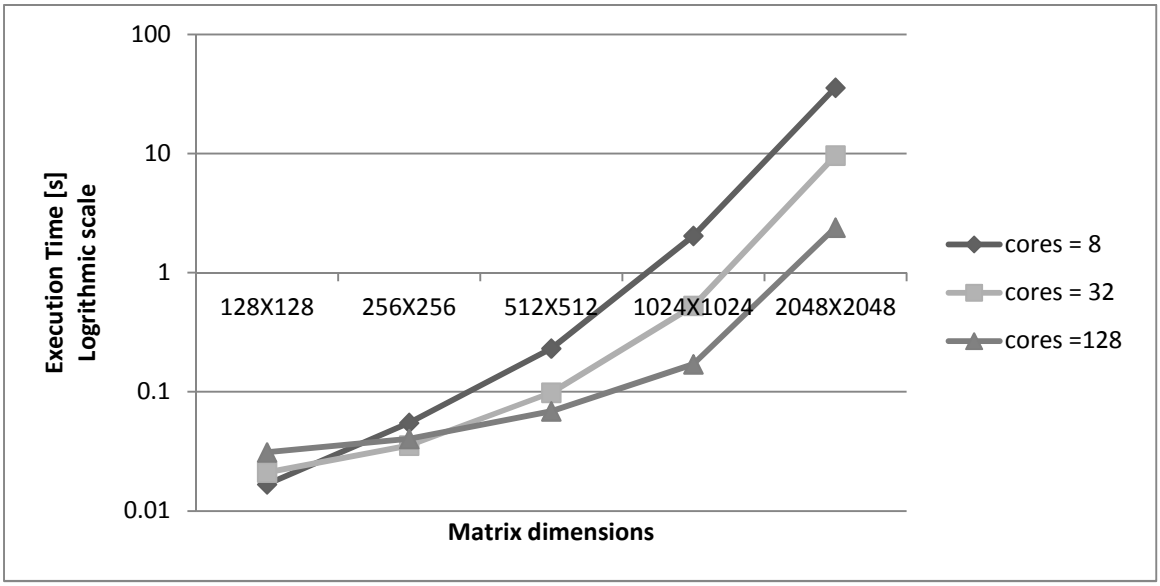
Call MPI_Cart_create (vu, …) // creating Cartesian topology.
Create row and columns groups as mentioned in Graph 5.8.
Every process initializes its assigned block.
Everyprocess gets required data using all-gather-all.
Everyprocess computes its data block as shown in Figure 5.7.

For (Traverse over all the required blocks – shaded ones in Figure 5.7)
Everyprocess computes its respective C block using OpenMP for loop parallelization.

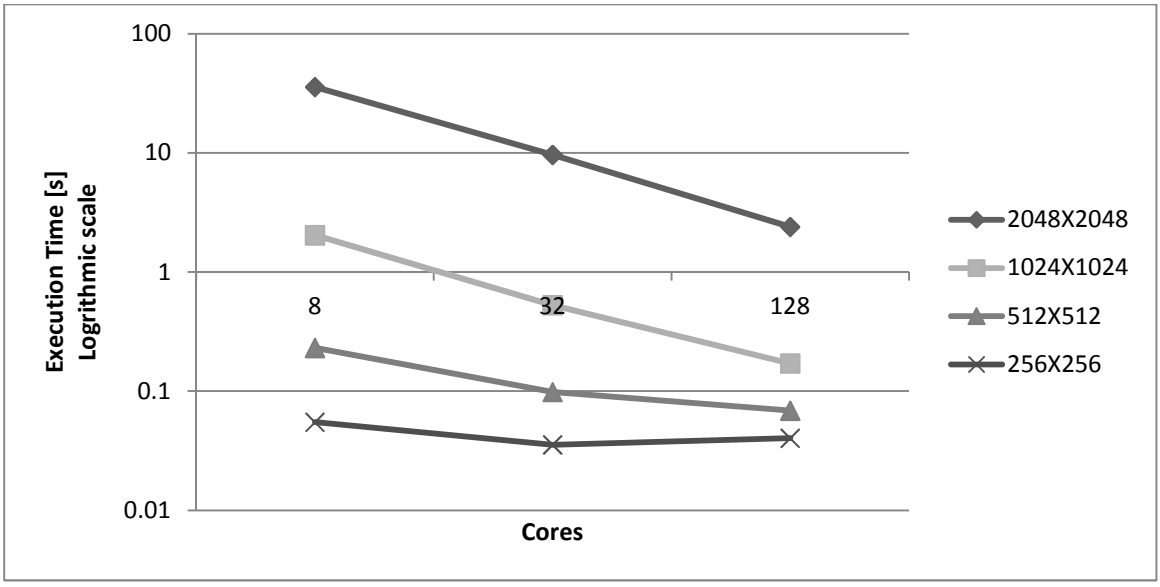
End for

Final result is collected using scheme mentioned in Figure 5.10.

Graph 5.18 shows execution time of various matrix sizes for 8, 32 and 128 cores. In the case of large number of cores i.e. 128,matrix multiplications for smaller matrix sizes (such as 128X128, 256X256, 512X512) takes more time to complete. Because in the case of smaller matrix sizes, data transferred through messages among the nodes is of smaller size and network is optimized for large data transfers. Graph 5.19 shows scalability graph, it can be noticed that for smaller matrix sizes it doesn't scales well because of smaller size message transfers.



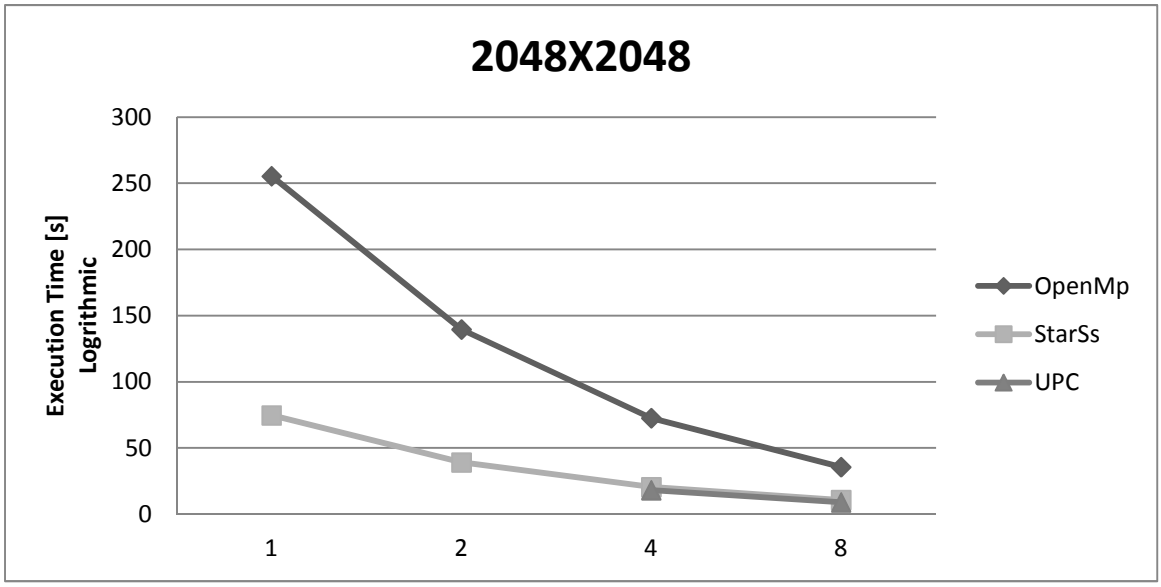
Graph 5.18: Execution time for various matrix sizes (all gather)



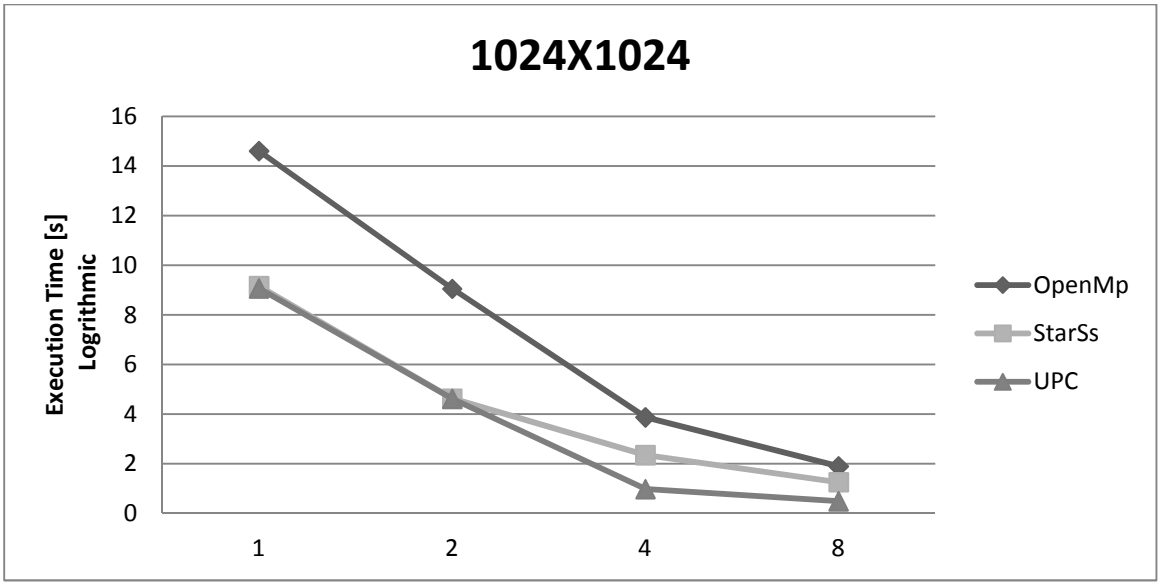
Graph 5.19: Scalability plot

6 Comparative performance

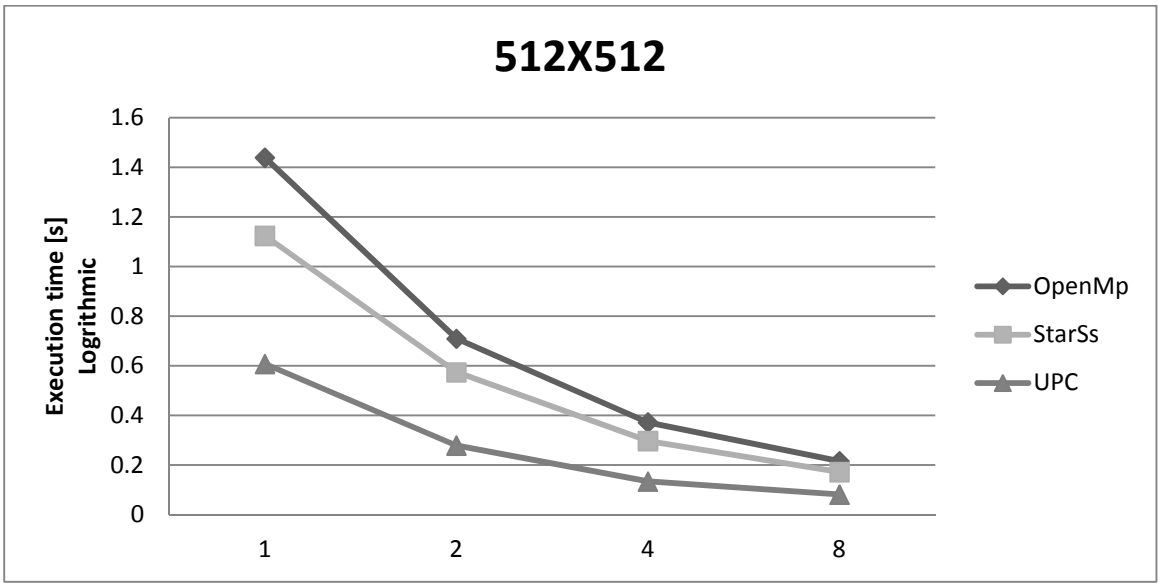
Graph 6.1, Graph 6.2and Graph 6.3 show shared memory comparison between OpenMP, StarSs and UPC for the matrix sizes of 2048X2048, 1024X1024 and 512X512. These tests are run on a Nehalem cluster node; details about the node can be seen in the start of section 5Implementation details and results. It can be seen that UPC performs better than the others for all matrix sizes. In Graph 6.3 UPC doesn't scale well, when cores are increased from 4 to the 8 as more time will be spent in moving around the data than computing the results. As mentioned in the start of section 5 Berkeley UPC implementation used has limit on block size. So, a point in the case of matrix dimensions 2048X2048 in Graph 6.1 and Graph 6.4 is missing.



Graph 6.1: Shared memory comparison for 2048X2048

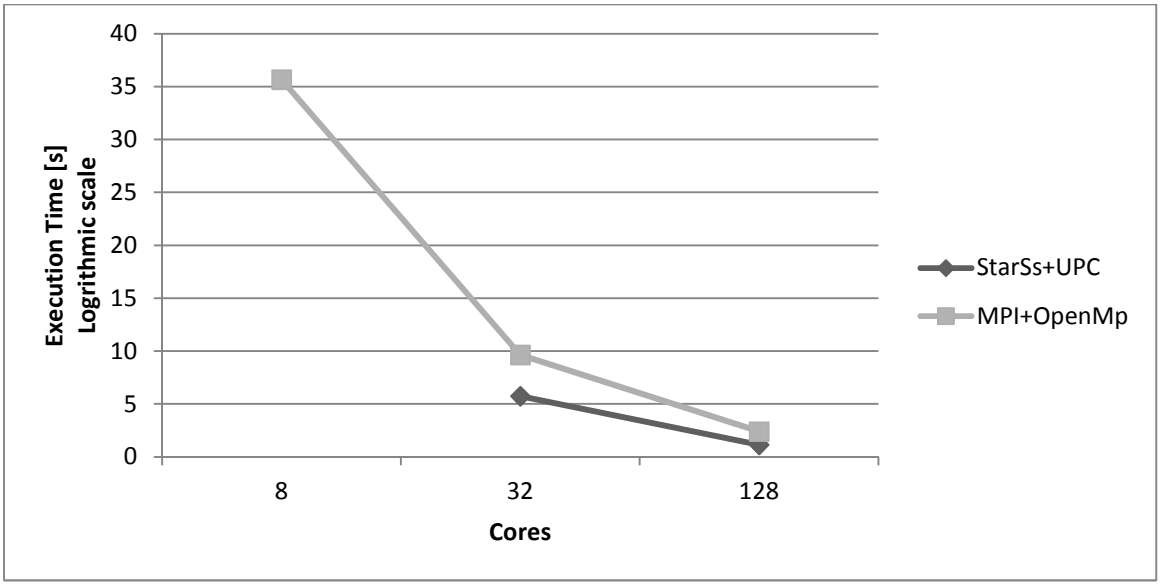


Graph 6.2: Shared memory comparison for 1024X1024

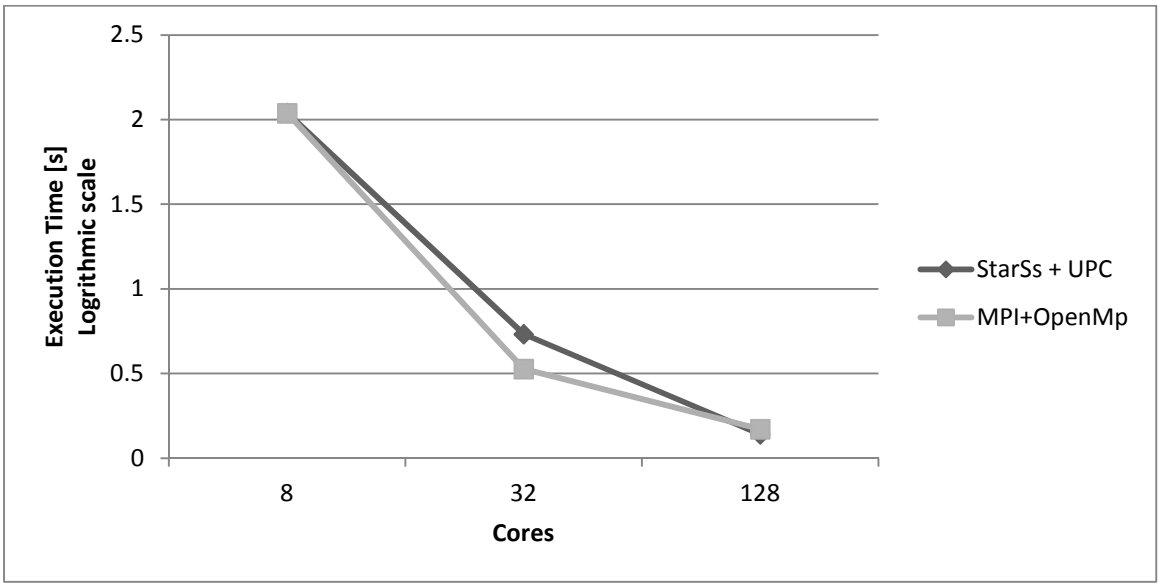


Graph 6.3: Shared memory comparison for 512X512

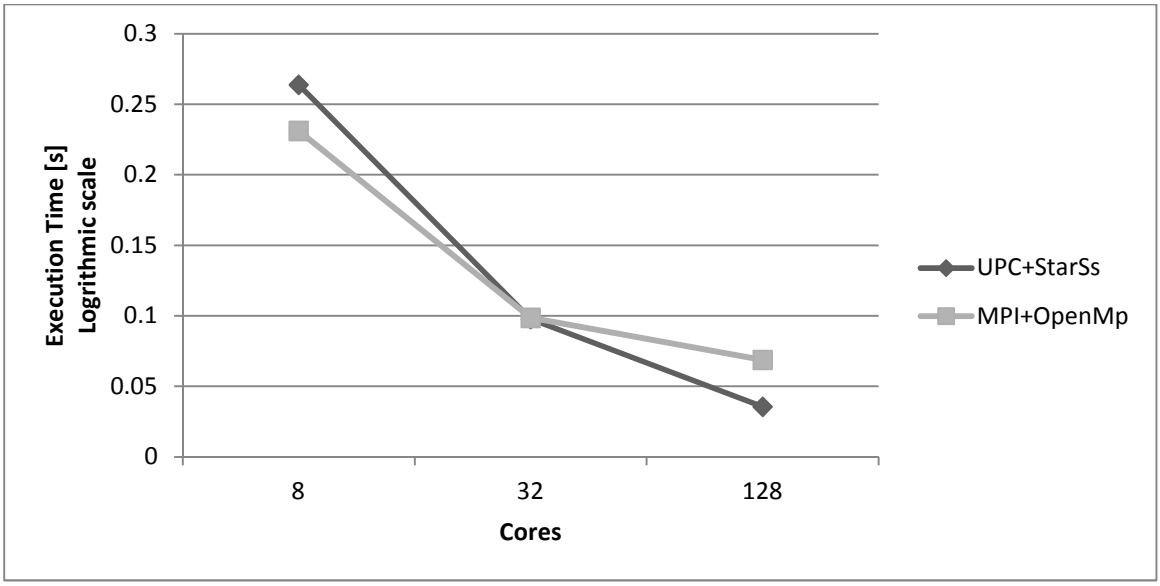
Graph 6.5 and Graph 6.6 shows comparison between hybrid memory systems. Legacy combination OpenMP/MPI (OpenMP for shared memory, MPI for distributed memory) is compared with StarSs/UPC (StarSs for shared memory, UPC for distributed memory) combination. Both of them show almost same results. It should be noted that even for small message transfers UPC + StarSs combination performs relatively better as compare to the MPI + OpenMP combination.



Graph 6.4: Hybrid (shared + distributed) memory comparison for 2048X2048 matrix



Graph 6.5: Hybrid (shared + distributed) memory comparison for 1024X1024 matrix



Graph 6.6: Hybrid (shared + distributed) memory comparison for 512X512 matrix

7 Discussion

MPI is the most common programming model used to write applications for distributed memory computing systems. In MPI user can fine tune their applications as they have complete control over data layout, communication, and load balance. Another advantage of MPI is portability i.e. developers don't have to rewrite application for new/different machines. One of the major drawbacks in using MPI is its difficulty in changing the sequential code to the parallel one.

UPC provides a way to solve this problem by retaining most of features of the sequential programming. It introduces the concept of global shared arrays which may be distributed over multiple nodes. In contrast to MPI, where one has to pack local data into the messages sharing with other nodes, remote data can be reached by accessing elements of the array. Therefore development time reduces. UPC also provides with the collectives communication call to support bulk data transfers as most communication architectures are optimized for bulk data transfers.

In many applications multi dimensional blocking is required to simplify the program structure. For example, take the case of the blocked matrix multiplication in Figure 5.3. The natural block-wise data distribution of the matrices cannot easily be implemented with UPC, as this supports only one-dimensional data distribution, not a two-dimensional as required. An alternative way needs to be devised for the distribution of the matrices using only UPC's one-dimensional blocking. This way, shown in Figure 5.5, increases the complexity of code.

As discussed in section 5.4, the use of virtual topologies in MPI simplifies the programming structure. If topology matches the underlying data distribution it improves the understandability of code which leads to less development time.

Because of the problems stated above developing SUMMA matrix multiplication for UPC takes considerable more time as compared to the MPI. As the lack of multidimensional blocking complicates the layout of the data (matrix storage in the memory) and lack of virtual topologies reduces the understandability of the code.

OpenMP is one of the most popular programming models used for shared memory parallelization. It consists of a collection of compiler directives, library routines, and environment variables that can be easily inserted into a sequential program to create a portable program that will run in parallel on shared memory architectures. It provides options for both task and data based parallelism. However, it is up to the user to ensure that performance does not suffer as a result of poor cache locality.

StarSs is designed for shared memory systems with specific attention paid to get a good cache performance. As discussed in section 3 it provides task based parallelism. So if our application is comprised of tasks, than using StarSs is the better way. If our problem requires data based parallelism, than using OpenMP will be the better choice.

Development of matrix multiplication algorithm, using StarSs and UPC generates comparable performance results as with OpenMP and MPI. But if we look at the development time required UPC and StarSs take longer time because of the reasons mentioned above. Therefore, it is right to say that UPC with features multidimensional blocking, virtual topology features and for applications which can be decomposed as tasks, StarSs + UPC combination will no doubt increase developers productivity.

8 Bibliography

[1] Open MPI: Open Source High Performance Computing. [Online]. http://www.open-mpi.org/

[2] OpenMP: The OpenMP® API specification for parallel programming. [Online]. http://openmp.org/wp/

[3] Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, Karen Warren William W. Carlson, "Introduction to UPC and Language Specification," May 13, 1999.

[4] Rosa M. Badia, Jesus Labarta Josep M. Perez, A Dependency-Aware Task-Based Programming Envoirment for Multi-Core Architectures, 2008.

[5] Steve Jackson Derek Hower, "TaskMan: Simple Task-Parallel Programming,".

[6] Alain Darte Guy-Rene Perrin, *The Data Parallel Programming Model: A Semantic Perspective*.

[7] D.B. Loveman and Maynard, MA Digital Equipment Corp., High performance Fortran, 1993.

[8] John Reid Robert W. Numrich, Co-array Fortran for parallel programming, 1998.

[9] "SMP superscalar (SMPSs) User's Manual," Barcelona, 2009.

[10] William Carlson, Thomas Sterling, Katherine Yelick Tarek El-Ghazawi, *Upc Distributed Shared Memory Programming*.

Appendix A

A.1. Graph data tables

Block Size	Matrix dimensions	Execution Time
32	128X128	0.0108
	256X256	0.046961
	512X512	0.212409
	1024X1024	1.979849
	2048X2048	34.016667
64	128X128	0.019785
	256X256	0.06277
	512X512	0.20896
	1024X1024	2.033456
	2048X2048	35.045333
128	128X128	0.027429
	256X256	0.115471
	512X512	0.400823
	1024X1024	2.057425
	2048X2048	33.726667
256	256X256	0.190936
	512X512	0.757126
	1024X1024	4.066777
	2048X2048	33.337

Table 8-1: Data for Graph 5.1

Matrix dimensions	Block Size	Execution Time
128X128	32	0.006777
	64	0.011809
	128	0.031536
256X256	32	0.038298
	64	0.04441
	128	0.059496
	256	0.192635
512X512	32	0.168071
	64	0.172193
	128	0.206418
	256	0.401219
1024X1024	32	1.232803
	64	1.259045
	128	1.299592
	256	1.910967
2048X2048	32	9.575
	64	10.717
	128	11.434
	256	12.616667

Table 8-2: Data for Graph 5.2

Matrix dimensions	row-wise distribution	block-wise distribution
128X128	0.011809	0.046961
256X256	0.04441	34.016667
512X512	0.172193	2.033456
1024X1024	1.259045	0.400823
2048X2048	10.717	0.757126

Table 8-3: Data for Graph 5.3

Matrix dimensions	cores	Execution Time
2048X2048	1	76.21
	2	39.567333
	4	20.387
	8	10.717
1024X1024	1	9.164759
	2	4.628079
	4	2.352953
	8	1.259045
512X512	1	1.123815
	2	0.573751
	4	0.297292
	8	0.172193

Table 8-4: Data for Graph 5.4

Matrix dimensions	cores	Execution Time
512X512	1	1.12349
	2	0.726774
	4	0.381239
	8	0.20896
1024X1024	1	9.201482
	2	15.064337
	4	4.124423
	8	2.033456
2048X2048	1	232.104667
	2	135.938333
	4	70.911333
	8	35.045333

Table 8-5: Data for Graph 5.5

UPC Threads	Matrix Dimensions	Execution Time
1	512X512	0.589
	1024X1024	9.057
4	512X512	0.139
	1024X1024	1.088
	2048X2048	19.61
16	512X512	0.046
	1024X1024	0.423
	2048X2048	8.484

Table 8-6: Data for Graph 5.6

UPC Threads	Matrix Dimensions	Execution Time
1	512X512	0.571333333
	1024X1024	10.25633333
4	512X512	0.182
	1024X1024	1.512333333
	2048X2048	24.3255952
16	512X512	0.585333333
	1024X1024	0.348611
	2048X2048	6.542146333
64	512X512	0.030975
	1024X1024	0.098644333
	2048X2048	0.739362333

Table 8-7: Data for Graph 5.7

Matrix Dimensions	UPC Threads	Execution Time
512X512	1	0.608
	2	0.279
	4	0.134
	8	0.081
1024X1024	1	9.069
	2	4.622
	4	0.985169
	8	0.494
2048X2048	4	18.241
	8	9.024

Table 8-8: Data for Graph 5.8

Cores	Matrix dimensions	Execution time
8	128X128	0.010966333
	256X256	0.047194667
	512X512	0.263851
	1024X1024	2.039269
32	128X128	0.007601667
	256X256	0.022133667
	512X512	0.097608667
	1024X1024	0.732649667
	2048X2048	5.744350333
128	128X128	0.010528667
	256X256	0.010528667
	512X512	0.035588
	1024X1024	0.141151
	2048X2048	1.144329333

Table 8-9: Data for Graph 5.9

Matrix dimensions	Cores	Execution time
512X512	8	0.263851
	32	0.097608667
	128	0.035588
1024X1024	8	2.039269
	32	0.732649667
	128	0.141151
2048X2048	32	5.744350333
	128	1.144329333

Table 8-10: Data for Graph 5.10

Cores	Matrix dimensions	Execution time
1	128X128	0.023921
	256X256	0.150875
	512X512	1.145826
	1024X1024	10.934753
	2048X2048	184.040243
4	128X128	0.009492
	256X256	0.050021
	512X512	0.408666
	1024X1024	3.213493
	2048X2048	29.454429
16	128X128	0.024115
	256X256	0.034381
	512X512	0.09383
	1024X1024	0.640018
	2048X2048	5.27758
64	128X128	0.066712
	256X256	0.068726
	512X512	0.090489
	1024X1024	0.199949
	2048X2048	1.723633

Table 8-11: Data for Graph 5.11

Matrix dimensions	Cores	Execution time
512X512	1	1.145826
	4	0.408666
	16	0.09383
	64	0.090489
1024X1024	1	10.934753
	4	3.213493
	16	0.640018
	64	0.199949
2048X2048	1	184.040243
	4	29.454429
	16	5.27758
	64	1.723633

Table 8-12: Data for Graph 5.12

Cores	Matrix dimensions	Execution time
1	128X128	0.028033
	256X256	0.149834
	512X512	1.292971
	1024X1024	12.073521
	2048X2048	202.51187
4	128X128	0.006765
	256X256	0.040909
	512X512	0.294649
	1024X1024	2.515457
	2048X2048	27.285554
16	128X128	0.015953
	256X256	0.035936
	512X512	0.088719
	1024X1024	0.57715
	2048X2048	4.587071
64	128X128	0.040586
	256X256	0.051999
	512X512	0.071215
	1024X1024	0.174
	2048X2048	1.141919

Table 8-13: Data for Graph 5.13

Matrix dimensions	Cores	Execution time
512X512	1	1.292971
	4	0.294649
	16	0.088719
	64	0.071215
1024X1024	1	12.073521
	4	2.515457
	16	0.57715
	64	0.174
2048X2048	1	202.51187
	4	27.285554
	16	4.587071
	64	1.141919

Table 8-14: Data for Graph 5.14

Matrix dimension	Cores	Execution time (all gather)	Execution time (non-blocking send/recv)
2048X2048	1	202.51187	184.040243
	4	27.285554	29.454429
	16	4.587071	5.27758
	64	1.141919	1.723633

Table 8-15: Data for Graph 5.15

Matrix dimensions	cores	Execution time
128X128	1	0.030218
	2	0.016432
	4	0.010132
	8	0.008193
256X256	1	0.180215
	2	0.097103
	4	0.055562
	8	0.044641
512X512	1	1.438879
	2	0.709328
	4	0.3719
	8	0.215667
1024X1024	1	14.604642
	2	9.051743
	4	3.87567
	8	1.90028
2048X2048	1	241.106132
	2	147.968762
	4	73.062732
	8	34.634772

Table 8-16: Data for Graph 5.16

Matrix dimensions	Cores	Execution time
512X512	1	1.438879
	2	0.709328
	4	0.3719
	8	0.215667
1024X1024	1	14.604642
	2	9.051743
	4	3.87567
	8	1.90028
2048X2048	1	241.106132
	2	147.968762
	4	73.062732
	8	34.634772

Table 8-17: Data for Graph 5.17

Cores	Matrix dimensions	Execution time
8	128X128	0.016755
	256X256	0.054899
	512X512	0.231178
	1024X1024	2.037397
	2048X2048	35.664118
32	128X128	0.021096
	256X256	0.035470333
	512X512	0.098574
	1024X1024	0.526931
	2048X2048	9.63201
128	128X128	0.031051667
	256X256	0.040283
	512X512	0.068761333
	1024X1024	0.170891667
	2048X2048	2.395041

Table 8-18: Data for Graph 5.18

Matrix dimensions	Cores	Execution time
512X512	8	0.231178
	32	0.098574
	128	0.068761333
1024X1024	8	2.037397
	32	0.526931
	128	0.170891667
2048X2048	8	35.664118
	32	9.63201
	128	2.395041

Table 8-19: Data for Graph 5.19

Matrix dimension	Cores	OpenMp	StarSs	UPC
	1	255.378054	74.69	
	2	139.550865	39.12	
	4	72.599632	20.51	18.251
2048X2048	8	35.639446	10.77	9.046

Table 8-20: Data for Graph 6.1

Matrix dimension	Cores	OpenMp	StarSs	UPC
	1	13.742839	10.17	9.15
	2	7.420541	5.178	4.625
	4	4.051131	2.623	0.959
1024X1024	8	1.873649	1.406	0.494

Table 8-21: Data for Graph 6.2

Matrix dimension	Cores	OpenMp	StarSs	UPC
	1	1.378361	1.23	0.59
	2	0.698123	0.628	0.285
	4	0.369102	0.325	0.138
512X512	8	0.215002	0.188	0.082

Table 8-22: Data for Graph 6.3

Matrix dimension	Cores	StarSs+UPC	MPI+OpenMp
	8		35.664118
2048X2048	32	5.74435	9.63201
	128	1.144329	2.395041

Table 8-23: Data for Graph 6.4

Matrix dimension	Cores	StarSs+UPC	MPI+OpenMp
1024X1024	8	2.039269	2.037397
	32	0.732649667	0.526931
	128	0.141151	0.170891667

Table 8-24: Data for Graph 6.5

Matrix dimension	Cores	StarSs+UPC	MPI+OpenMp
512X512	8	0.263851	0.231178
	32	0.097608667	0.098574
	128	0.035588	0.068761333

Table 8-25: Data for Graph 6.6

Appendix B

B.1. StarSs row-wise matrix multiplication code

```
//Writtern By Muhammad Wahaj Sethi
//Email muhammad.wahaj@gmail.com
//Following program multiplies two matrix using starSs
//Matrix date breakdown for initialization: Each StarSs thread has
assigned
//a block to work on. BlockSize parameter decides dimension of block
//Data breakdown for computation: Matrix is partitioned among threads
//blockwise.
//Dimensions of block: blockSize X blockSize
//Each thread has to work on submatrix of size blockSize X blockSize

#include "stdio.h"
#include "string.h"
#include "stdlib.h"
#include "time.h"
#include "math.h"
#include "rdtsc.h"

#define blockSize 4
#define ARows 8
#define ACols 8
#define BRows 8
#define BCols 8
#define blkInCols (BCols / blockSize)

int ARowsP, AColsP, BRowsP, BColsP;

//Print complete matrix including padded values.
void printMatriceDebug (double A [], char matName)
{
      int i;
      int rows, cols;

      if (matName == 'A')
      {
            rows = ARowsP;
            cols = AColsP;
      }
```

```
	else if (matName == 'B')
	{
		rows = BRowsP;
		cols = BColsP;
	}

	else if (matName == 'C')
	{
		rows = ARowsP;
		cols = BColsP;
	}

	else
		printf ("Function printMatrice: Wrong Matrix name!! Valid
values 'A', 'B' and 'C' \n");

	for (i = 0; i < (rows * cols); i++)
	{
		printf ("%f ", A [i]);

		if (((i + 1) % cols) == 0)
			printf ("\n");
	}
}

//Input matrice start address and matrice name which can only be A, B
or C.
//Function: Prints actual matrix only.
void printMatrice (double A [], char matName)
{
	int i, j;
	int rows, rowsP, cols, colsP;
	double *temp = &A [0];

	if (matName == 'A')
	{
		rows = ARows;
		rowsP = ARowsP;
		cols = ACols;
		colsP = AColsP;
	}

	else if (matName == 'B')
	{
		rows = BRows;
		rowsP = BRowsP;
		cols = BCols;
		colsP = BColsP;
	}

	else if (matName == 'C')
	{
		rows = ARows;
		rowsP = ARowsP;
		cols = BCols;
		colsP = BColsP;
	}

	else
```

```
            printf ("Function printMatrice: Wrong Matrix name!! Valid
values 'A', 'B' and 'C' \n");


      //Prints matrice here. Just print out actual elements not padded
ones.
      for (i = 0; i < (rowsP * colsP); i++)
      {
            //printf ("%i:%i:%i:%i:%i  ", i, i%colsP, i/colsP, (i+1) %
colsP, (i / colsP));    //i % colsP tells column number
            //i % rowsP tells row number
            //if both of above within matrix dim print the element
            //else do nothing.
            if (((i % colsP) < cols) && ((i / colsP) < rows))
            {
                  j = 0;
                  printf ("%f ", A [i]);
            }

            //keep tracks of row. If next element is not on same row
print next line character.
            //((i / rowsP) < rows) condition make sure that no newline
character when row exceds
            //actual dimension.
            if ((((i+1) % colsP) == 0) && ((i / colsP) < rows))
                  printf ("\n");
      }

      printf ("\n");
}

#pragma css task output (subMat)
void initWithZero (double *subMat)
{
      memset (&subMat [0], blockSize * blockSize * sizeof (double),
0);
}

#pragma css task input(rowLimit, colLimit, rows, cols) output(subMat)
void initWithRand (double *subMat, int rowLimit, int colLimit, int
rows, int cols)
{
      int i, j;

      //matrix initilazation here. Actual matrix entries have assigned
some random value
      //and padding bit is set to 0.
      //rowLimit and colLimit variable are used to differentiate
between actual and padding entries.

      for (i = 0; i < blockSize; i++)
      {
            for (j = 0; j < blockSize; j++)
            {
                  if ((j < colLimit) && (i < rowLimit))
                  {
                        subMat [i * cols + j] = 1;
                  }

                  else
                  {
```

```
                        subMat [i * cols + j] = 0;
                  }
            }
      }

}

#pragma css task input(subMatA , subMatB) inout(subMatC)
void multiply (double *subMatA, double *subMatB, double *subMatC)
{
      int row, col, i;

      for (row = 0; row < blockSize; row++)
      {
            for (col = 0; col < blockSize; col++)
            {

                  for (i = 0 ; i < blockSize; i++)
                  {
                        subMatC [row * BColsP + col] += subMatA [row *
AColsP + i] * subMatB [col + i * BColsP];
                  }
            }
      }
}

//compares two values and returns minimum.
int min (int a , int b)
{

      if ( a < b)
            return a;

      else
            return b;
}

int main ()
{
      unsigned long long clk0, clk1;
        double timeDiff, timeDiff1;
        time_t t0, t1;
      int k = 0;

      if (ACols != BRows) //Checking matrice dimension
      {
            printf ("Matrice dimension doesn't matches.\n");
            exit (0);
      }

      //loop index variables
      int i, j;

      //Determining how many blocks present in matrices dimensions.
When dimensions are not multiple of blockSize adding an
      //extra block.
      int blkARows = ceil (((double) ARows / blockSize));
      int blkACols = ceil (((double) ACols / blockSize));
      int blkBRows = ceil (((double) BRows / blockSize));
      int blkBCols = ceil (((double) BCols / blockSize));
```

```
	//These variables are used to note program start and end time.
	clock_t start, end, diff;


	//used for storing size of matrices.
	int sizeA, sizeB, sizeC;

	sizeA = blkARows * blkACols * blockSize * blockSize;
	sizeB = blkBRows * blkBCols * blockSize * blockSize;
	sizeC = blkARows * blkBCols * blockSize * blockSize;

	//Used for matrice storage.
	double A [sizeA]; //Creating Matrice A
	double B [sizeB]; //Creating Matrice B
	double C [sizeC]; //Creating Matrice C

	//variable used to store size of ARows, ACols, BRows, BCols
after padding
	ARowsP = blkARows * blockSize;
	AColsP = blkACols * blockSize;
	BRowsP = blkBRows * blockSize;
	BColsP = blkBCols * blockSize;

	//use in initilazation phase to store upper limits.
	int colLimit, rowLimit;

	#pragma css start
	start = clock ();
	  t0 = time (NULL);
	  clk0 = rdtsc ();

	////////////////////////////////////////////////////////////////
/////
	//This portion initialize matrices.
	//Matrice is divided between different tasks depending on blocks
of rows present in a matrice.

	for (i = 0; i < blkARows; i++)
	{
		for (j = 0;j < blkACols; j++)
		{
			//first calculating upper limits of block. Then
comparing it with matrix dimension
			//and chossing minimum. After that subtracting
starting pos of block to get relative
			//displacement inside a block
			colLimit = min (j * blockSize + blockSize, ACols) -
j * blockSize;
			rowLimit = min (i * blockSize + blockSize, ARows) -
i * blockSize;

			initWithRand (&A [i * blockSize * AColsP + j *
blockSize], rowLimit, colLimit, ARowsP, AColsP);

		}
	}

	for (i = 0; i < blkBRows; i++)
	{
		for (j = 0;j < blkBCols; j++)
		{
```

```
                  colLimit = min (j * blockSize + blockSize, BCols) -
j * blockSize;
                  rowLimit = min (i * blockSize + blockSize, BRows) -
i * blockSize;
                  initWithRand (&B [i * blockSize * BColsP + j *
blockSize], rowLimit, colLimit, BRowsP, BColsP);
            }
      }

      for (i = 0; i < blkARows; i++)
      {
            for (j = 0; j < blkBCols; j++)
            {
                  initWithZero (&C [i * blockSize * BColsP + j *
blockSize]);
            }
      }

      ////////////////////////////////////////////////////////////////
/////


      for (i = 0; i < blkARows; i++)
      {
            for (j = 0;j < blkBCols; j++)
            {
                  //Getting to appropriate A row's index. First at 0
then at multiple of block
                  //Getting to appropriate B col's position
                  //Check rough sheet for detail example
                  //C [i * AColsP * blockSize + j * blockSize]
                  //i * AColsP * blockSize determines row displacement
                  //blkACols * blockSize tells about total elements in
a row of C
                  //blockSize here tells in how many rows a block
occupies

                  for (k = 0; k < blkInCols; k++)
                  {
      multiply (&A [k * blockSize + i * AColsP * blockSize], &B [k *
BColsP * blockSize + j * blockSize], &C[i * BColsP * blockSize + j *
blockSize]);
                  }
            }
      }

      #pragma css barrier
        end = clock ();
        t1 = time (NULL);
        clk1 = rdtsc ();

      #pragma css finish
        //stop cache stats gather here
        diff = end - start;
        printf ("CPU_Time_taken: %6.6f\n", (double) diff /
CLOCKS_PER_SEC);
        printf ("Wall_Time_taken: %ld\n", (long) (t1 - t0));
        printf ("Wall_Time_taken_rdtsc %f \n", (clk1 -clk0) / (2.8 *
1e9));
```

```
}
```

B.2. StarSs block-wise matrix multiplication code

```
//Writtern By Muhammad Wahaj Sethi
//Email muhammad.wahaj@gmail.com
//Following program multiplies two matrix using starSs
//Matrix date breakdown for initialization: Each StarSs thread has
//assigned
//number of rows to work on. BlockSize parameter decides number of
//rows to be allocated.
//Data breakdown for computation: Each thread has assigned number of
//rows to work on.
//blockSize parameter used to determine number of rows allocated per
//thread.


#include "stdio.h"
#include "string.h"
#include "stdlib.h"
#include "time.h"
#include "math.h"
#include "rdtsc.h"

#define blockSize 32
#define ARows 2048
#define ACols 2048
#define BRows 2048
#define BCols 2048

int ARowsP, AColsP, BRowsP, BColsP;

void printMatriceDebug (double A [], char matName)
{
      int i;
      int rows, cols;

      if (matName == 'A')
      {
            rows = ARowsP;
            cols = AColsP;
```

```
	}

	else if (matName == 'B')
	{
		rows = BRowsP;
		cols = BColsP;
	}

	else if (matName == 'C')
	{
		rows = ARowsP;
		cols = BColsP;
	}

	else
		printf ("Function printMatrice: Wrong Matrix name!! Valid
values 'A', 'B' and 'C' \n");

	for (i = 0; i < (rows * cols); i++)
	{
		printf ("%f ", A [i]);

		if (((i + 1) % cols) == 0)
			printf ("\n");
	}
}

//Input matrice start address and matrice name which can only be A, B
or C.
//Function: Prints actual matrix only.
void printMatrice (double A [], char matName)
{
	int i, j;
	int rows, rowsP, cols, colsP;
	double *temp = &A [0];

	if (matName == 'A')
	{
		rows = ARows;
		rowsP = ARowsP;
```

```
            cols = ACols;
            colsP = AColsP;
      }

      else if (matName == 'B')
      {
            rows = BRows;
            rowsP = BRowsP;
            cols = BCols;
            colsP = BColsP;
      }

      else if (matName == 'C')
      {
            rows = ARows;
            rowsP = ARowsP;
            cols = BCols;
            colsP = BColsP;
      }

      else
            printf ("Function printMatrice: Wrong Matrix name!! Valid
values 'A', 'B' and 'C' \n");



      //Prints matrice here. Just print out actual elements not padded
ones.
      for (i = 0; i < (rowsP * colsP); i++)
      {
            //i % colsP tells column number
            //i % rowsP tells row number
            //if both of above within matrix dim print the element
            //else do nothing.

            if (((i % colsP) < cols) && ((i / colsP) < rows))
            {
                  j = 0;
                  printf ("%f ", A [i]);
            }
```

```
            //keep tracks of row. If next element is not on same row
print next line character.
            //((i / rowsP) < rows) condition make sure that no newline
character when row exceds
            //actual dimension.

            if ((((i+1) % colsP) == 0) && ((i / colsP) < rows))
                  printf ("\n");
      }

      printf ("\n");
}

#pragma css task input(test, mat, task_no) output(subMat)
void initWithRand (double *subMat, char mat, int test, int task_no)
{
      int i, j;
      int cols, colsP, rows;

      if (mat == 'A')
      {
            cols = ACols;
            colsP = AColsP;
            rows = ARows;
      }

      else if (mat == 'B')
      {
            cols = BCols;
            colsP = BColsP;
            rows = BRows;
      }

      else
            printf ("Function printMatrice: Wrong Matrix name!! Valid
values 'A', 'B' and 'C' \n");

      //matrix initilazation here. Actual matrix entries have assigned
some random value
      //and padding bit is set to 0.
      //j < temp condition takes care of col limit
```

```
	//((task_no * blockSize + i) < temp) takes care of row limit.

	for (i = 0; i < blockSize; i++)
	{
		for (j = 0; j < colsP; j++)
		{
			if ((j < cols) && ((task_no * blockSize + i) < 
rows))
			{
				subMat [i * colsP + j] = 1;
			}

			else
			{
				subMat [i * colsP + j] = 0;
			}
		}
	}

}

#pragma css task input(subMatA , subMatB, rowLimit) output(subMatC)
void multiply (double *subMatA, double *subMatB, double *subMatC, int 
rowLimit)
{
	int row, col, i;

	for (row = 0; row < rowLimit; row++)
	{
		for (col = 0; col < BColsP; col++)
		{
			subMatC [row * BColsP + col] = 0;

			for (i = 0 ; i < AColsP; i++)
			{
				subMatC [row * BColsP + col] += subMatA [row * 
AColsP + i] * subMatB [col + i * BColsP];
			}
		}
	}
}
```

```
int min (int a, int b)
{
      if ( a < b)
            return a;

      else
            return b;
}

int main ()
{
      if (ACols != BRows) //Checking matrice dimension
      {
            printf ("Matrice dimension doesn't matches.\n");
            exit (0);
      }

      //loop index variables
      int i, j;
      time_t t0, t1; //used for wall time.
      unsigned long long clk0, clk1, diff;
      double timeDiff, timeDiff1;

      //Determining how many blocks present in matrices dimensions.
When dimensions are not multiple of blockSize adding an
      //extra block.
      int blkARows = ceil (((double) ARows / blockSize));
      int blkACols = ceil (((double) ACols / blockSize));
      int blkBRows = ceil (((double) BRows / blockSize));
      int blkBCols = ceil (((double) BCols / blockSize));

      //These variables are used to note program start and end time.
      clock_t start, end, diffMine, diffMkl; //used for cpu time.



      //used for storing size of matrices.
      int sizeA, sizeB, sizeC;
```

```
	sizeA = blkARows * blkACols * blockSize * blockSize;
	sizeB = blkBRows * blkBCols * blockSize * blockSize;
	sizeC = blkARows * blkBCols * blockSize * blockSize;

	//Used for matrice storage.
	double A [sizeA]; //Creating Matrice A
	double B [sizeB]; //Creating Matrice B
	double CMine [sizeC]; //Creating Matrice C
	double CMkl [sizeC];



	//variable used to store size of ARows, ACols, BRows, BCols
after padding
	ARowsP = blkARows * blockSize;
	AColsP = blkACols * blockSize;
	BRowsP = blkBRows * blockSize;
	BColsP = blkBCols * blockSize;

	int rowLimit;

	#pragma css start
	////////////////////////////////////////////////////////////////
/////
	//This portion initialize matrices.
	//Matrice is divided between different tasks depending on blocks
of rows present in a matrice.

	start = clock ();
	t0 = time (NULL);
	clk0 = rdtsc ();

	for (i = 0; i < blkARows; i++)
	{
		initWithRand (&A [i * blockSize * AColsP], 'A', 1, i);
	}

	for (i = 0; i < blkBRows; i++)
	{
		initWithRand (&B [i * blockSize * BColsP], 'B', 1, i);
	}
```

```
      ////////////////////////////////////////////////////////////////
/////
      # pragma css barrier

      for (i = 0; i < blkARows; i++)
      {
                  //Getting to appropriate A row's index. First at 0
then at multiple of block
                  //Getting to appropriate B col's position
                  //Check rough sheet for detail example
                  //C [i * AColsP * blockSize + j * blockSize]
                  //i * AColsP * blockSize determines row displacement
                  //blkACols * blockSize tells about total elements in
a row of C
                  //blockSize here tells in how many rows a block
occupies

                  rowLimit = min (i * blockSize + blockSize, ARows) -
i * blockSize;
                  multiply (&A [i * AColsP * blockSize], &B [0],
&CMine[i * BColsP * blockSize], rowLimit);
      }

      #pragma css barrier
      end = clock ();
      t1 = time (NULL);
      clk1 = rdtsc ();
      #pragma css finish
      diffMine = end - start;
      timeDiff1 = (clk1 - clk0) / (2.8 * 1e9);

      printf ("CPU_Time_taken: %6.6f \n", (double) diffMine /
CLOCKS_PER_SEC);
      printf ("Wall_Time_taken: %ld \n", (long) (t1 - t0));
      printf ("Wall_Time_taken_rdtsc %f \n", timeDiff1);

      printf ("Matrice A %i X %i...\n", ARows, ACols);
      printMatrice (A, 'A');

      printf ("Matrice B %i X %i...\n", BRows, BCols);
      printMatrice (B, 'B');
```

```
      printf ("Matrice C %i X %i...\n", ARows, BCols);
      printMatrice (CMine, 'C');

}
```

B.3. StarSs + UPC matrix multiplication code

B.3.1. starSs.h

```
#ifndef __starSs_h__
#define __starSs_h__

#include "hybrid.h"

#pragma css task input (blkCmpl, MYTHREAD, mat)
void checkAllBlksInit (int *blkCmpl, int MYTHREAD, char mat);

void masterThread (int MYTHREAD, double * aPtr, double *bPtr, double
*cPtr);

#pragma css task input(A, B, MYTHREAD, row, col, AFlg, BFlg) inout(C)
void mul (double *A, double *B, double *C, int MYTHREAD, int row, int
col, int *AFlg, int *BFlg);

#pragma css task input(MYTHREAD, start) output(ptr, blkCmpl)
void initRand (double *ptr, int MYTHREAD, int start, int *blkCmpl);

#pragma css task input(MYTHREAD, start) output(ptr)
void initZero (double *ptr, int MYTHREAD, int start);

#pragma css task input(blkLoc, locA) output(strLoc, AFlg) highpriority
void copyRmtBlkA (double *strLoc, int blkLoc, int locA, int *AFlg);

#pragma css task input (blkLoc, locB) output (strLoc, BFlg)
highpriority
void copyRmtBlkB (double *strLoc, int blkLoc, int locB, int *BFlg);
```

```
#endif
```

B.3.2. starSs.c

```
#include "hybrid.h"
#include "stdio.h"
#include "stdlib.h"
#include "starSs.h"
#include "string.h"

      //This function calls mul with relavent index depending on
thread number.
      //For details on data decomposition see read me file.
      void masterThread (int MYTHREAD, double *aPtr, double *bPtr,
double *cPtr)
      {
            int i, j, k, tempA, tempB, tempC, temp, temp4Col,
temp4Row, l, m;
            double *tempAPtr, *tempBPtr, *tempCPtr;
            double *rmtBlkAArr, *rmtBlkBArr;
      int l2ABlks [l2BlkInRows * l2BlkInCols];
      int l2BBlks [l2BlkInRows * l2BlkInCols];
      int locA, locB; //Contains location of remote block.
      int ACopiedFlg [l1BlkInRows], BCopiedFlg [l1BlkInRows];


      //Extra storage used can be removed here. only l1BlkInRows space
required.
      rmtBlkAArr = (double *) malloc (l1BlkSize * l1BlkInRows * sizeof
(double));
      rmtBlkBArr = (double *) malloc (l1BlkSize * l1BlkInRows * sizeof
(double));

      memset ((void *) &l2ABlks [0], 0, sizeof (int) * l2BlkInRows *
l2BlkInCols);
      memset ((void *) &l2BBlks [0], 0, sizeof (int) * l2BlkInRows *
l2BlkInCols);
      memset ((void *) &ACopiedFlg [0], 1, sizeof (int) *
l1BlkInRows);
      memset ((void *) &BCopiedFlg [0], 1, sizeof (int) *
l1BlkInRows);

      #pragma css start

      for (i = 0; i < l2BlkInRows; i++)
      {
            for (j = 0; j < l2BlkInCols; j++)
            {
                  initRand (&aPtr [j * l2Cols + i * l2BlkSize *
l2BlkInCols], MYTHREAD, MYTHREAD * l1BlkSize + i * l2BlkInCols *
l2BlkSize + j * l2Cols, &l2ABlks [i * l2BlkInCols + j]);
            }
      }

      for (i = 0; i < l2BlkInRows; i++)
```

```
      {
            for (j = 0; j < l2BlkInCols; j++)
            {
                  initRand (&bPtr [j * l2Cols + i * l2BlkSize *
l2BlkInCols], MYTHREAD, MYTHREAD * l1BlkSize + i * l2BlkInCols *
l2BlkSize + j * l2Cols, &l2BBlks [i * l2BlkInCols + j]);
            }
      }

      for (i = 0; i < l2BlkInRows; i++)
      {
            for (j = 0; j < l2BlkInCols; j++)
            {
            initZero (&cPtr [j * l2Cols + i * l2BlkSize *
l2BlkInCols], MYTHREAD, MYTHREAD * l1BlkSize + i * l2BlkInCols *
l2BlkSize + j * l2Cols);
            }
      }


      checkAllBlksInit (&l2ABlks [0] , MYTHREAD, 'A');
      checkAllBlksInit (&l2BBlks [0] , MYTHREAD, 'B');



      //perform calculations untill all blocks computed.
      for (k = 0; k < l1BlkInRows ; k++)
      {
            //exactly one l1 block in each col. True when respective A
blocal present in local memory.
            //condition used to avoid calculation of already
calculated block.

            locA = (MYTHREAD / l1BlkInCols) * l1BlkInCols + k;
            locB = (MYTHREAD % l1BlkInCols) + k * l1BlkInCols;

            if (((k%l1BlkInCols) == (MYTHREAD%l1BlkInCols)) && (k ==
(MYTHREAD/l1BlkInCols)))
            {
                  tempAPtr = &aPtr [0];
                  tempBPtr = &bPtr [0];
                  ACopiedFlg [k] = 1;
                  BCopiedFlg [k] = 1;
            }

            else if ((k%l1BlkInCols) == (MYTHREAD%l1BlkInCols))
            {
                  tempAPtr = &aPtr [0];
                  tempB = (k * l1BlkInCols + (MYTHREAD % l1BlkInCols))
* l1BlkSize;
                  ACopiedFlg [k] = 1;
                  copyRmtBlkB (&rmtBlkBArr [k * l1BlkSize], tempB,
locB, &BCopiedFlg [k]);
                  tempBPtr = &rmtBlkBArr [k * l1BlkSize];
            }

            //Exactly one l1 block in each row. True when respective B
block present in local memory.
            else if (k == (MYTHREAD/l1BlkInRows))
            {
                  tempBPtr = &bPtr [0];
```

```
                  tempA = k * l1BlkSize + (MYTHREAD / l1BlkInCols) *
l1BlkInCols * l1BlkSize;
                  copyRmtBlkA (&rmtBlkAArr [k * l1BlkSize], tempA,
locA, &ACopiedFlg [k]);
                  BCopiedFlg [k] = 1;
                  tempAPtr = &rmtBlkAArr [k * l1BlkSize];
            }

            //Cols blocks taken care of by first part of expression
            //Row block will be taken care of by second part of
expression.
            //Expression second part first determines block row number
            //than Xly it with number of blocks in a col to get block
number
            //and atlast its Xlied by number elements in a block to
get appropriate array
            //index.
            else if (!((k%l1BlkInCols) == (MYTHREAD%l1BlkInCols)) &&
!(k == (MYTHREAD/l1BlkInRows)))
            {
                  tempA = k  * l1BlkSize + (MYTHREAD / l1BlkInCols) *
l1BlkInCols * l1BlkSize;
                  tempB = (k * l1BlkInCols + (MYTHREAD % l1BlkInCols))
* l1BlkSize;
                  copyRmtBlkA (&rmtBlkAArr [k * l1BlkSize], tempA,
locA, &ACopiedFlg [k]);
                  copyRmtBlkB (&rmtBlkBArr [k * l1BlkSize], tempB,
locB, &BCopiedFlg [k]);
                  tempAPtr = &rmtBlkAArr [k * l1BlkSize];
                  tempBPtr = &rmtBlkBArr [k * l1BlkSize];
            }


            for (l = 0; l < l2BlkInRows; l++)
            {
                  for (m = 0; m < l2BlkInCols; m++)
                  {
                              for (j = 0; j < l2BlkInRows; j++)
                              {

mul (&tempAPtr[l * l2Rows * l1Rows + j * l2Cols] , &tempBPtr [j *
l2Rows * l1Cols + m * l2Cols], &cPtr [l * l2Rows * l1Cols + m *
l2Cols], MYTHREAD, l, m, &ACopiedFlg [k], &BCopiedFlg [k]);
                              }
                  }
            }
      }

      #pragma css finish


      free (rmtBlkAArr);
      free (rmtBlkBArr);
}

#pragma css task input (blkCmpl, MYTHREAD, mat)
void checkAllBlksInit (int *blkCmpl, int MYTHREAD , char mat)
{
      int temp, i;
      void (*funcPtr) (int) = NULL;
```

```
	if (mat == 'A')
		funcPtr = &setFlagA;

	else
		funcPtr = &setFlagB;

	while (1)
	{
		temp = 0;

		for (i = 0; i < (l2BlkInCols * l2BlkInRows); i++)
		{
			if (blkCmpl [i] == 0)
				temp = 1;
		}

		if (temp == 0)
		{
			(*funcPtr) (MYTHREAD);
			return;
		}

	}
}

#pragma css task input(A, B, MYTHREAD, row, col, AFlg, BFlg) inout (C)
void mul (double *A, double *B, double *C, int MYTHREAD, int row, int
col, int *AFlg, int *BFlg)
{
	int i, j, k,temp;

	for (i = 0; i < l2Rows; i++)
	{
		for (j = 0; j < l2Cols; j++)
		{
			for (k = 0; k < l2Rows; k++)
			{
				temp = i * l1Cols + j;
				C [temp] += A [i * l1Cols + k] * B [j + l1Cols
* k];
			}
		}
	}
}

#pragma css task input (MYTHREAD, start) output(ptr, blkCmpl)
void initRand (double *ptr, int MYTHREAD, int start, int *blkCmpl)
{
	int i, j;

	for (i = 0; i < l2Rows; i++)
	{
		for (j = 0; j < l2Cols ; j++)
			ptr [i * l1Cols + j] = 1;
	}

	blkCmpl [0] = 1;
}

#pragma css task input (MYTHREAD, start) output(ptr)
void initZero (double *ptr, int MYTHREAD, int start)
```

```
{
      int i, j;

      for (i = 0; i < l2Rows; i++)
      {
            for (j = 0; j < l2Cols; j++)
                  ptr [i * l1Cols + j] = 0;
      }
}

#pragma css task input (blkLoc, locA) output (strLoc, AFlg)
highpriority
void copyRmtBlkA (double *strLoc, int blkLoc, int locA, int *AFlg)
{
      while (getFlagA (locA) == 0);
      copyRemoteBlockA (strLoc, blkLoc);
}

#pragma css task input (blkLoc, locB) output (strLoc, BFlg)
highpriority
void copyRmtBlkB (double *strLoc, int blkLoc, int locB, int *BFlg)
{
      while (getFlagB (locB) == 0);
      copyRemoteBlockB (strLoc, blkLoc);
}
```

B.3.3. hybrid.h

```
#ifndef __hybrid_h__
#define __hybrid_h__


//Mandatory conditions for level1 (l1) and level2 (l2) blk conditions.
//l2 <= l1 and l2 should be multiple of l1
//l1 <= (dimensions of matrices) and should be multiple of matrice
dimensions.


#include "starSs.h"
#define ARows 2048
#define ACols 2048
#define BRows 2048
#define BCols 2048
#define l1Rows 1024
#define l1Cols 1024
#define l2Rows 1024
#define l2Cols 1024
#define l1BlkSize (l1Rows * l1Cols)
#define l1BlkInRows (ARows / l1Rows)
```

```
#define l1BlkInCols (ACols / l1Cols)
#define l2BlkSize (l2Rows * l2Cols)
#define l2BlkInRows (l1Rows / l2Rows)
#define l2BlkInCols (l1Cols / l2Cols)

void memGet (double *ptr);
void barrier ();
void copyRemoteBlockA (double *APtr, int ALoc);
void copyRemoteBlockB (double *BPtr, int BLoc);
int getFlagA (int loc);
int getFlagB (int loc);
void setFlagA (int loc);
void setFlagB (int loc);

#endif
```

B.3.4. hybrid.upc

```
#include "hybrid.h"
#include "upc_relaxed.h"
#include "time.h"
#include "unistd.h"
#include "rdtsc.h"

shared [l1BlkSize] double a [ARows * ACols];
shared [l1BlkSize] double b [BRows * BCols];
shared [l1BlkSize] double c [ARows * BCols];
shared [1] int blkFlagsA [l1Rows * l1Cols];
shared [1] int blkFlagsB [l1Rows * l1Cols];

int getFlagA (int loc)
{
      return blkFlagsA [loc];
}

int getFlagB (int loc)
{
      return blkFlagsB [loc];
}

void setFlagA (int loc)
{
      blkFlagsA [loc] = 1;
}

void setFlagB (int loc)
{
      blkFlagsB [loc] = 1;
}

void barrier ()
```

```
{
      upc_barrier;
}

void copyRemoteBlockA (double *APtr, int ALoc)
{
      int i;
      upc_memget (APtr, &a [ALoc], l1BlkSize * sizeof (double));
}

void copyRemoteBlockB (double *BPtr, int BLoc)
{
      int i;
      upc_memget (BPtr, &b [BLoc], l1BlkSize * sizeof (double));
}

void printArray (char mat)
{
      int i, j, k, l, m, n;

      for (l = 0; l < l1BlkInRows; l++)
      {
            for (i = 0; i < l1Rows; i++)
            {
                  for (j = 0; j < l1BlkInCols; j++)
                  {
                        for (k = 0; k < l1Cols; k++)
                        {
                              if (mat == 'A')
                              {
                                    printf ("%6.2f ", a [k + j *
l1BlkSize + i * l1Cols + l * l1BlkSize * l1BlkInCols]);
                              }

                              else if (mat == 'B')
                              {
                                    printf ("%6.2f ", b [k + j *
l1BlkSize + i * l1Rows + l * l1BlkSize * l1BlkInCols]);
                              }

                              else if (mat == 'C')
                              {
                                    printf ("%6.2f ", c [k + j *
l1BlkSize + i * l1Rows + l * l1BlkSize * l1BlkInCols]);
                              }
                        }

                  }

                  printf ("\n");
            }
      }
}


int main ()
{
      int i, j;
      double *aPtr, *bPtr, *cPtr;
      clock_t start, end, diff;
      unsigned long long clk0, clk1;
```

```
	double timeDiff;

	aPtr = (double *) &a [MYTHREAD * l1BlkSize];
	bPtr = (double *) &b [MYTHREAD * l1BlkSize];
	cPtr = (double *) &c [MYTHREAD * l1BlkSize];

	upc_barrier;
	printf ("here\n");

	if (MYTHREAD == 0)
	{
		start = clock (); //Noting start time of computation.
		clk0 = rdtsc ();
	}

	masterThread (MYTHREAD, aPtr, bPtr, cPtr);
	upc_barrier;

	if (MYTHREAD == 0)
	{
		end = clock (); //Noting end time of computation.
		clk1 = rdtsc ();
	}

	if (MYTHREAD == 0)
	{
		diff = end - start;
		timeDiff = (clk1 - clk0) / (2.8 * 1e9);
		printf ("Time_taken ... %6.6f \n", (double) diff /
CLOCKS_PER_SEC);
		printf ("Time_taken_rdtsc ... %6.6f \n", timeDiff);

/*		printf ("Matrice A  %i X %i ... \n", ARows, ACols);
		printArray ('A');

		printf ("Matrice B  %i X %i ... \n", BRows, BCols);
		printArray ('B');

		printf ("Matrice C  %i X %i ... \n", ARows, BCols);
		printArray ('C');
*/
	}

	return 0;
}
```

Erklärung

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

Unterschrift:

Stuttgart,