International Journal of Soft Computing and Engineering (IJSCE)
ISSN: 2231-2307, Volume-4 Issue-2, May 2014

# Heterogeneous Parallel Programming

**Rahul Ravindran, Riya Suchdev, Yash Tanna**

***Abstract*—This paper presents Heterogeneous Parallel Computing (HPC), which is a well-orchestrated and co-ordinated effective use of a suite of diverse high performance machines to provide super-speed processing for computationally demanding tasks with diverse computational needs. GPUs are accoutered with a much more throughput oriented design as compared to that of the CPUs thus making them a powerful alternative to boast overall performance. It is now used all the way from mobile computing to supercomputing, like in Blue Star Super Computers. Upcoming Exascale and Petascale systems have embraced even heterogeneity in order to overcome power limitations. This paper also illustrates programming example using CUDA C to demonstrate the efficiency achieved in problems like matrix multiplication using a more heterogeneous approach as compared to that of sequential approach. It also explains how Heterogeneous Parallel Programming is a plausible, novel technique which allows to exploit inherent capabilities of a wide range of computational machines to solve computationally intensive problems that have several types of embedded parallelism by breaking it into separate modules. This paper also puts light on the challenges and concerns which exist when programming in HPC environment and some techniques to alleviate them.***

***Index Terms*—About four key words or phrases in alphabetical order, separated by commas.**

## I. INTRODUCTION

Heterogeneity has emerged as a prevalent characteristic of parallel computing platforms. Heterogeneous parallel platforms combine processing units (PUs) that have different instruction set architectures or different micro-architectures, e.g., multi- core processors, general-purpose graphics processing units (GPGPUs) and other many-core accelerators [1].The PUs that constitute a heterogeneous platform are effectively optimized to serve different workload characteristics (e.g., latency-sensitive vs. throughput-sensitive, coarse-grained vs. fine-grained parallel, etc.). Studies have argued for the fundamental benefits of heterogeneity in parallel computing and demonstrated speedups on heterogeneous platforms for a wide range of application domains. However, programming heterogeneous parallel platforms is commonly cited as a major challenge that must be addressed before their potential can be realized for mainstream computing. Progress has been made in raising the level of abstraction at which many-core accelerators GPUs are programmed—from low-level, domain-specific APIs to high-level programming languages.

**Manuscript Received on May 2014.**

**Rahul Ravindran**, Computer Engineering, Mumbai University, India.
**Riya Suchdev**, Computer Engineering, Mumbai University, Mumbai, India.
**Yash Tanna**, Computer Engineering, Mumbai University, Mumbai, India.

While they cover much ground, these frameworks still leave the programmer with the significant challenges of tuning the accelerator code for performance, partitioning, mapping and scheduling an application on the different PUs of a heterogeneous platform, and managing the data transfers between their (often distinct) memory hierarchies. Recent research efforts address the challenge of providing the programmer with a unified view of the ensemble of PUs and their memory hierarchies on the one hand, while on the other hand achieving good performance and performance portability.

Conventional homogeneous systems usually use one mode of parallelism in a given machine like Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), or vector processing and thus cannot adequately meet the requirements of applications that require more than one type of parallelism. As a result, any single type of machine often spends its time executing code for which it is poorly suited. Moreover, many applications need to process information at more than one level concurrently, with different types of parallelism at each level. Thus there rises a need to use a system that gives much more throughput which would give much more efficiency to the already existent system.

Our main objective is to study the various aspects of such an environment and also show improvements with the help of statistical data that we have established by performing the simple task of matrix multiplication using CUDA C. Heterogeneous parallel programming helps us to gather effectively use the graphical processing unit in order to achieve maximum throughput my using multiple cores i.e. threading effectively.

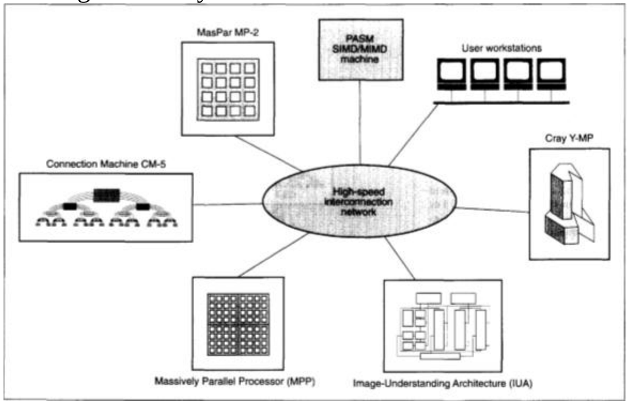

MasPar MP-2
PASM SIMD/MIMD machine
User workstations
Cray Y-MP
Connection Machine CM-5
High-speed interconnection network
Massively Parallel Processor (MPP)
Image-Understanding Architecture (IUA)

**Fig. I Heterogeneous Computing Environment**

## II. NEED FOR HETEROGENEOUS PARALLEL PROGRAMMING

Conventional homogeneous systems usually use one mode of parallelism in a given machine like SIMD,

MIMD or vector processing and thus cannot adequately meet the requirements of applications that require more than one type of parallelism. As a result, any single type of machine often spends its time executing code for which it is poorly suited.
Moreover, many applications need to process information at more than one level concurrently, with different types of parallelism at each level [2]. For such applications, users of a conventional multiprocessor system must either settle for degraded performance on the existing hardware or acquire more powerful (and expensive) machines.
Each type of homogeneous system suffers from inherent limitations. If the data distribution of an application and the resulting computations cannot exploit these features, the performance degrades severely.
The quest for higher computational power suitable for a wide range of applications at a reasonable cost has exposed several inherent limitations of homogeneous systems. Replacing such systems with yet more powerful homogeneous systems is not feasible. Case reports highlight how your research contributes to the current knowledge in the field and mention the next steps or what remains. Feel free to explain why your results falsify current theories if that is the case. Make sure that your discussion is concise and informative. If you ramble and include a great deal of unnecessary information, your paper will likely get rejected or at least be looked upon less favorably.

## III. IMPLEMENTATION

We consider two approaches to using the HC paradigm. The first one analyzes an application to explore embedded heterogeneous parallelism. Researchers must devise new algorithms or modify existing ones to exploit the heterogeneity present in the application. Based on these algorithms, users develop the code to be executed by the machines. In the second approach, an existing parallel code of the application is taken as input. To run this code in an HC environment, users must profile the types of heterogeneous parallelism embedded in the code. For this purpose, code-type profilers need to be designed. Figures 3 and 4 illustrate these approaches. However, both approaches need strategies for partitioning, mapping, scheduling, and synchronization. New tools and metrics for performance evaluation are also required. [3]

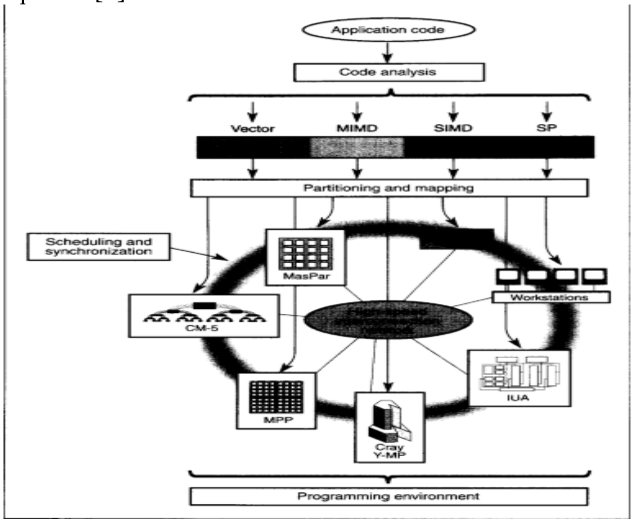


**Fig. II Compiler Directed Approach**

Algorithm design. Heterogeneous computing opens new opportunities for developing parallel algorithms. In this section, we identify the efforts needed to devise suitable algorithms. The following issues must be considered by the designer:

(1) The types of machines available and their inherent computing characteristics,
(2) Alternate solutions to various sub problems of the application, and
(3) The costs of performing the communication over the network. [4]

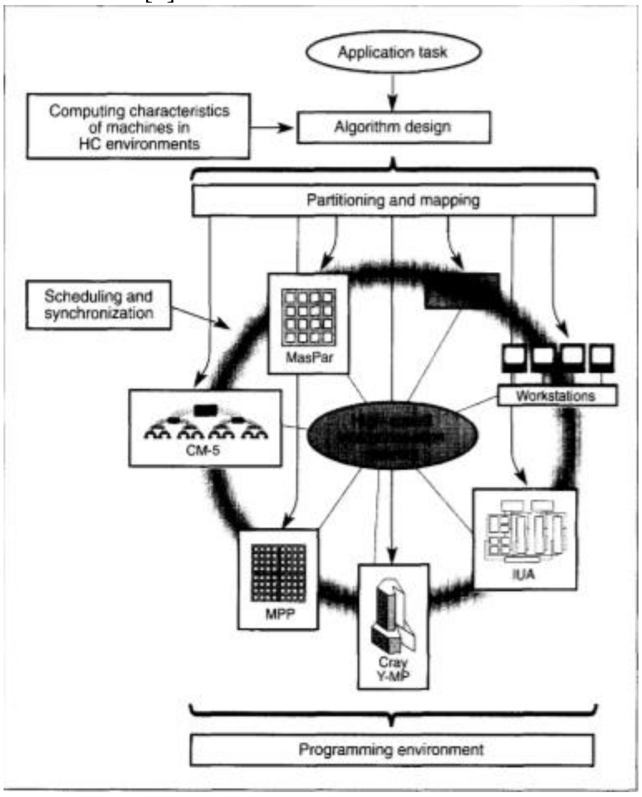


**Fig. III User Directed Approach**

Computation in HC can be classified into two types

**1) Metacomputing:**

Computations in this class fall into the category of coarse - grained heterogeneity. Instructions belonging to a particular class of parallelism are grouped to form a module; each module is then executed on a suitable parallel machine. Metacomputing refers to heterogeneity at the module level.

**2) Mixed-mode computing:**

In this fine grained heterogeneity, almost every alternate parallel instruction belongs to a different class of parallel computation. Programs exhibiting this type of heterogeneity are not suitable for execution on a suite of heterogeneous machines because the communication overhead due to frequent exchange of information between machines can become a bottleneck. However, these programs can be executed efficiently on a single machine such as PASM (Partitionable SIMD/MIMD) which incorporates heterogeneous modes of computation. Mixed-mode computing refers to heterogeneity
at the instruction level..

## IV. IMPLEMENTATION USING PROGRAMMING

**CUDA** (formerly **Compute Unified Device Architecture**) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphical processing units (GPUs) that they produce. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVidia GPUs become accessible for computation like CPUs. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general-purpose (i.e., not exclusively graphics) problems on GPUs is known as GPGPU. CUDA provides both a low level API and a higher level API. The initial CUDA SDK was made public on 15 February 2007, for Microsoft Windows and Linux. Mac OS X support was later added in version 2.0, which supersedes the beta released February 14, 2008. CUDA works with all **NVidia** GPUs from the G8x series onwards, including GeForce, Quadra and the Tesla line. CUDA is compatible with most standard operating systems.

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs:

- Scattered reads – code can read from arbitrary addresses in memory
- Shared memory – CUDA exposes a fast shared memory region (up to 48KB per Multi-Processor) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and read backs to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups

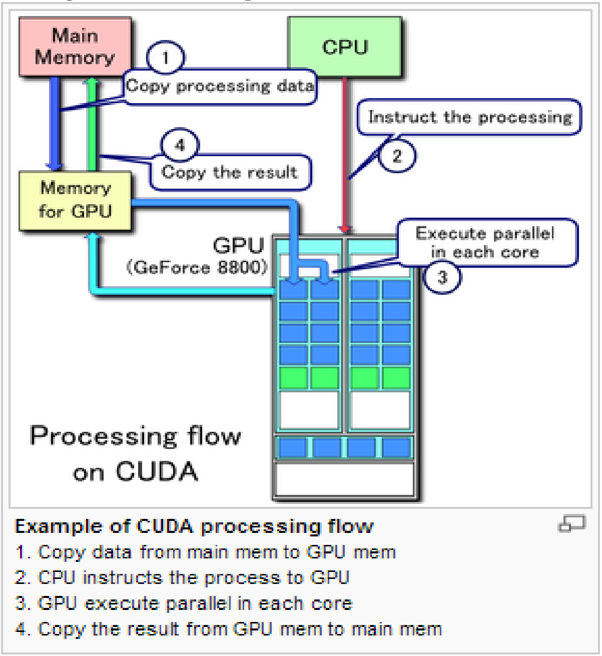
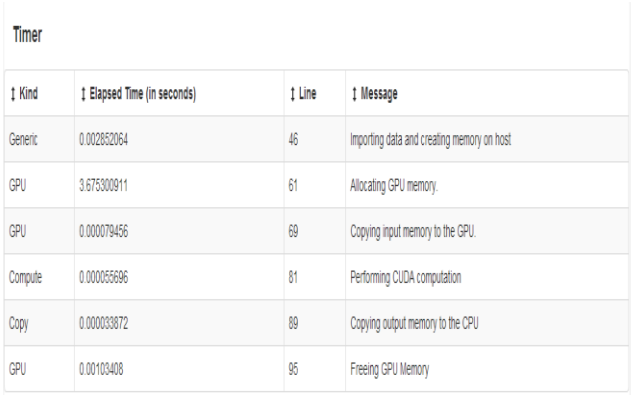

Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU (GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA
Example of CUDA processing flow
1. Copy data from main mem to GPU mem
2. CPU instructs the process to GPU
3. GPU execute parallel in each core
4. Copy the result from GPU mem to main mem

**Fig. IV Processing Flow of CUDA**

### A. *Using CUDA to solve Matrix Multiplication Problems*

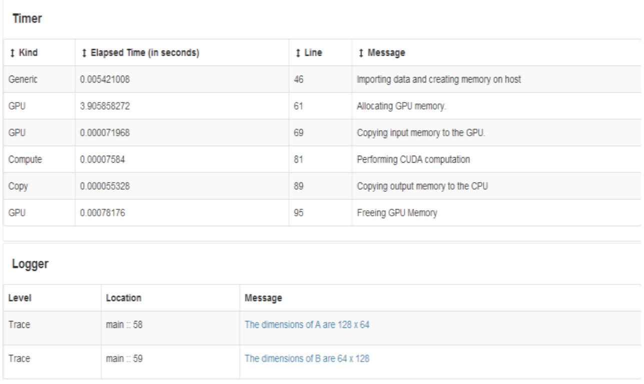


Timer

| ↕ Kind | ↕ Elapsed Time (in seconds) | ↕ Line | ↕ Message |
|---|---|---|---|
| Generic | 0.002852064 | 46 | Importing data and creating memory on host |
| GPU | 3.675300911 | 61 | Allocating GPU memory. |
| GPU | 0.000079456 | 69 | Copying input memory to the GPU. |
| Compute | 0.000055696 | 81 | Performing CUDA computation |
| Copy | 0.000033872 | 89 | Copying output memory to the CPU |
| GPU | 0.00103408 | 95 | Freeing GPU Memory |

**Fig. V Result of Matrix Multiplication 64X64**

The figure above shows the execution time elapsed for a matrix multiplication operation which involved two matrices of the size 64X64.As shown in the figure above it only took 0.000055696 sec to perform the matrix multiplication. Thus using the GPU i.e. **NVIDIA** CUDA processor a 10 fold efficiency is achieved in the case of matrix multiplication problem, the average time required to solve such a problem using a traditional language like java is very high. Thus programming in CUDA C i.e. using the GPU gives a great efficiency in computation



Timer

| ↕ Kind | ↕ Elapsed Time (in seconds) | ↕ Line | ↕ Message |
|---|---|---|---|
| Generic | 0.005421008 | 46 | Importing data and creating memory on host |
| GPU | 3.905858272 | 61 | Allocating GPU memory. |
| GPU | 0.000071968 | 69 | Copying input memory to the GPU. |
| Compute | 0.00007584 | 81 | Performing CUDA computation |
| Copy | 0.000055328 | 89 | Copying output memory to the CPU |
| GPU | 0.00078176 | 95 | Freeing GPU Memory |

Logger

| Level | Location | Message |
|---|---|---|
| Trace | main :: 58 | The dimensions of A are 128 x 64 |
| Trace | main :: 59 | The dimensions of B are 64 x 128 |

**Fig. VI Result of Matrix Multiplication 128X64**

Computation time for a matrix multiplication problem of size 128X64 with a matrix of size 64X128 takes a computation time of 0.00007854 sec.

### B. Performance measurement

There are various methods that are used to measure the performance of a certain parallel program. No single method is usually preferred over another since each of them, as will be seen later on, reflects certain properties of the parallel code.

#### *1) Speedup*

In the simplest of terms, the most obvious benefit of using a parallel computer is the reduction in the running time of the code. Therefore, a straightforward measure of the parallel performance would be the ratio of the execution time on a single processor (the sequential version) to that on a multicomputer. This ratio is defined as the speedup factor and is given as

$$S(n) = \frac{\text{Execution time using one processor}}{\text{Execution time using N processors}} = \frac{t_s}{t_n}$$

where $t_s$ is the execution time on a single processor and $t_n$ is the execution time on a parallel computer.

S (n) therefore describes the scalability of the system as the number of processors is increased. The ideal speedup is n when using n processors, i.e. when the computations can be divided into equal duration processes with each process running on one processor (with no communication overhead). Ironically, this is called embarrassingly parallel computing!

In some cases, super linear speedup (S (n)>n) may be encountered. Usually this is caused by either using a suboptimal sequential algorithm or some unique specification of the hardware architecture that favours the parallel computation. For example, one common reason for super linear speedup is the extra memory in the multiprocessor system. The speedup of any parallel computing environment obeys the Amdahl's Law. Amdahl's law states that if *F* is the fraction of a calculation that is sequential (i.e. cannot benefit from parallelisation), and (1 − *F*) is the fraction that can be parallelised, then the maximum speedup that can be achieved by using *N* processors is

$$\frac{1}{F + (1-F)/N}.$$

In the limit, as *N* tends to infinity, the maximum speedup tends to 1/*F*. In practice, price/performance ratio falls rapidly as *N* is increased once (1 − *F*)/*N* is small compared to *F*.As an example, if *F* is only 10%, the problem can be sped up by only a maximum of a factor of 10, no matter how large the value of *N* used. For this reason, **parallel computing** is only useful for either small numbers of processors, or problems with very low values of *F*: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce *F* to the smallest possible value.

*2) Efficiency*

The efficiency of a parallel system describes the fraction of the time that is being used by the processors for a given computation. It is defined as

$$E(n) = \frac{\text{Execution time using one processor}}{\text{Execution time using N processors x N}} = \frac{t_s}{Nt_n}$$

which yields the following $E(n) = \frac{S(n)}{N}$

for example, if E = 50%, the processors are being used half of the time to perform the actual computation.

*3) Cost*

The cost of a computation in a parallel environment is defined as the product of the number of processors used times the total execution time

$$cost = Nt_n$$

The above equation can be written as a function of the efficiency by using the fact that $t_p = \frac{t_n}{S(n)}$ *which yields*

$$cost = \frac{Nt_s}{S(n)} = \frac{t_s}{E(n)}$$

### *C. Limitations of CUDA*

- Texture rendering is not supported (CUDA 3.2 and up addresses this by introducing "surface writes" to CUDA arrays, the underlying opaque data structure).
- Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency (this can be partly alleviated with asynchronous memory transfers, handled by the GPU's DMA engine)
- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task (e.g. traversing a space partitioning data structure during ray tracing).
- Unlike OpenGL, CUDA-enabled GPUs are only available from NVidia
- Valid C/C++ may sometimes be flagged and prevent compilation due to optimization techniques the compiler is required to employ to use limited resources.

CUDA (with compute capability 1.x) uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.

## V. CHALLENGES

- **Machine Selection:** An interesting problem appears in design HPP environment is to find the most appropriate suite of heterogeneous machines for a given collection of application tasks subject to a given constraint such as cost, execution time etc.
- With the rise of HPP, software cost has been increasing more than hardware costs.

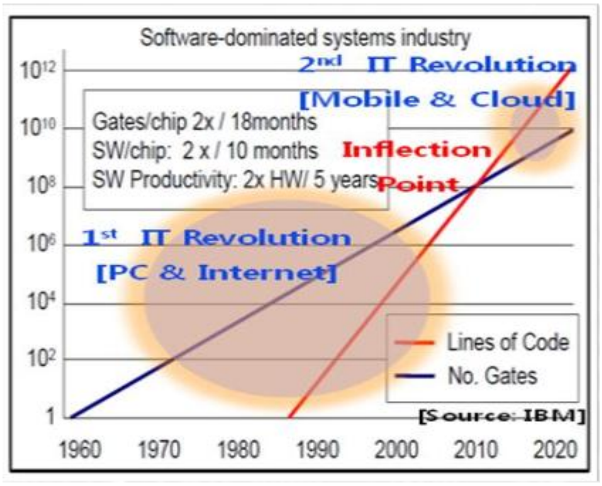


**Fig. VII Depicting rise in software costs**

- Task synchronization and scheduling becomes more challenging in comparison to homogenous systems.
- Inter-connectivity amongst heterogeneous machines adds additional challenges.
- Applications may run faster or slower on multi-core processors depending on the needs of the application.
- Multicore processors can hurt performance when applications:

1) Cannot break apart their operations well because much information must be held in memory at once.
2) Have many users doing the same thing at the same time, such as accessing a database. [6]

## VI. OVERCOMMING THE CHALLENGES

- Code profiling is used to categorize portions of application code.
- Analytical Benchmarking is used to determine appropriate machines.

It permits researchers to determine the relative effectiveness of a given parallel machine on various types of computation. Some experimental results obtained analytical benchmarking show that SIMD machines are well-suited for operations such as matrix computations and low-level image processing. Several solution have been proposed for code mapping, such as ***Cluster-M.*** [7]

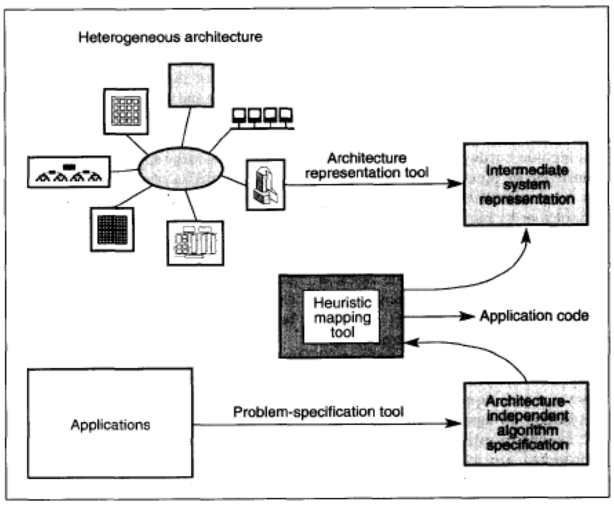

Heterogeneous architecture
Architecture representation tool
Intermediate system representation
Heuristic mapping tool
Application code
Applications
Problem-specification tool
Architecture-independent algorithm specification

**Fig. VIII Cluster-M based heuristic mapping methodology**

- Feund has proposed the **Optimal Selection Theory (OST)** to choose an optimal configuration of machines for executing an application task on a heterogeneous suite of computers with the assumption that the number of machines available is unlimited. It is also assumed that machines matching the given set of code types are available and that the application code is decomposed into equal-sized modules.
- The Parallel Virtual Machine (PVM) is a software tool for parallel network of computers.

PVM enables users to exploit their existing computer hardware to solve much larger problems at less additional cost. It is designed to allow a network of heterogeneous UNIX and/or Windows machines to be used as a single distributed parallel processor.

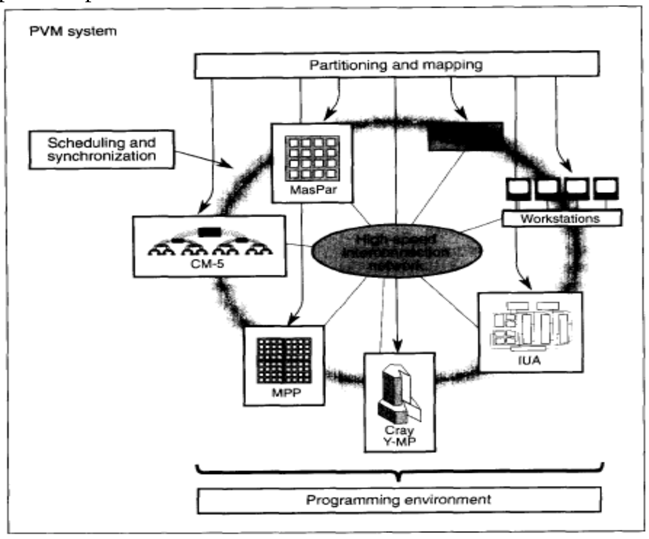


**Fig. IX An overview of the Parallel Virtual Machine system**

## VII. WHAT LIES AHEAD

Hybrid-core computing is the technique of extending a commodity instruction set architecture (e.g. x86) with application-specific instructions to accelerate application performance It is a form of heterogeneous computing wherein asymmetric computational units coexist with a "commodity" processor. Hybrid-core processing differs from general heterogeneous computing in that the computational units share a common logical address space, and an executable is composed of a single instruction stream—in essence a contemporary coprocessor the instruction set of a hybrid-core computing system contains instructions that can be dispatched either to the host instruction set or to the application-specific hardware. Typically, hybrid-core computing is best deployed where the predominance of computational cycles are spent in a few identifiable kernels, as is often seen in high-performance computing applications. Acceleration is especially pronounced when the kernel's logic maps poorly to a sequence of commodity processor instructions, and/or maps well to the application-specific hardware. [8]

## VIII. PUBLICATION PRINCIPLES

Computer systems will change in significantly the coming decade and beyond. Although steadily improving compiler technology will enable programmers to target more and more different architectures using the same high-level source code, there will always be important accelerators with little or no sophisticated compiler sup- port that require expert-created low-level modules. Enabling the easy integration of different programming models and different processors, and the efficient reuse of expert-developed code will be key to navigating this on-going transition. In this paper we have presented a pragmatic approach to use throughput devices i.e. GPU cores to effectively increase performance, our aim is to not use just the CPU but use the CPU in sequential programming operations and whenever there is a need for operations involving parallelism to make use of the GPU. GPUs can be 10+X faster in such parallel parts and we highlight these facts in the implementation using CUDA C, where a simple problem of matrix multiplication is made faster by effectively using the GPU cores. The CUDA threads consists of arrays of thread which when initialized with the matrices can perform operation of matrix multiplication in a pipelined fashion leading to a great speed up in the performance of the machine. Hybrid-core computing is used to accelerate applications beyond what is currently physically possible with off-the-shelf processors, or to lower power & cooling costs in a data center by reducing computational footprint. Thus to make use of CPU during sequential operations and the GPU during parallel operations helps to achieve more throughput than a standard machine that relies only on the CPU for its operation.

## REFERENCES

1. Jacques A. Piennar,Srimat Chakradar and Anand Ragunathan "automatic generation of software peipeline for heterogeneous parallel systems"
2. Ashfaq A. Khokar "Heterogeneous Computing:Challenges and opportunities"
3. T. Berg and H.J. Siegel, "Instruction Execution Trade-offs for SIMD vs. MIMD vs. Mixed-Mode Parallelism,'' Proc. Int'l Parallel Processing Symposium (IPPS), IEEE CS Press. Los Alamitos. Calif., Order NO. 2167. 1991, pp. 301-308.
4. A. Khokhar et al.. "Heterogeneous Supercomputing: Problems and Issues," Proc. Workshop on Heterogeneous Processing, IEEE CS Press, Los Alamitos. California Order No. 2702. 1992. pp. 3-12.
5. R. Freund. "Optimal Selection Theory for Superconcurrency." Proc. 89 Super- computing, IEEE CS Press, Los Alamitos, Calif., Order No. M2021 (microfiche), 1989. pp. 13-17.
6. The Multi-core Dilemma white paper by CITO Research
7. Heterogeneous supercomputing: Problems and issues Ashfaq Khokhar, Viktor Prasanna, Muhammad Shaaban, Cho-Li Wang
8. Instruction set innovations for the convey HC-1 by TM Brewer

## AUTHORS PROFILE

**Rahul Ravindran** is a third year Bachelors of engineering student in VESIT, Mumbai-University of Mumbai.

**Riya Suchdev** is a third year Bachelors of engineering student in VESIT, Mumbai-University of Mumbai.

**Yash Tanna** Y is a third year Bachelors of engineering student in VESIT, Mumbai-University of Mumbai.