iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 3 **Issue:** IX **Month of publication:** September 2015

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)

Design of a Parallel Multi-Threaded Programming Model for Multicore Architecture with Resource Sharing

Ajit Giri[1], Padmapriya Patil[2]
[1] M.Tech,[2] Professor, Department of Electronics and Communication Engineering,
Poojya Doddappa Appa College of engineering,
Gulbarga, Karnataka, India.

Abstract- Multi-core architectures have become main stream, and multi-core processors are found in products ranging from small portable cell phones to large computer servers. In parallel, research on real-time systems has mainly focused on traditional single-core processors. Hence, in order for real-time systems to fully leverage on the extra capacity offered by multi-core processor, different design techniques, scheduling approaches, and real-time analysis methods have to be developed. With the arrival of Multi- core Processors (MPs), every processor has now built-in parallel computational power and that can be fully utilized only if the program in execution is written accordingly. Also existing memory system and parallel developments tools do not provide adequate support for general purpose multi-core programming and unable to utilize all available cores efficiently. This study is an attempt to come up with some solutions for the challenges that multi-core processing is currently facing. This project contributes by developing a new multi-thread parallel programming model, "Spmd" (Single program multiple data programming model), for multi-core processor. The SPMD is a serial-like task-oriented parallel programming model which consists of a set of rules for algorithm decomposition and a library of primitives to exploit thread-level parallelism and concurrency on multi-core processors. The programming model works equally well for different classes of problems including basic, complex, regular and irregular problems. Its parallel execution makes it more efficient and less time consuming and its large set of input parameters also provides a wide range of simulation scenarios. Hence in this project the work deals with the synchronization procedure each application is assumed to be allocated on one dedicated core. However, it is further extended the synchronization procedure to be applicable for applications allocated on multiple dedicated cores of a multi-core platform. Likewise, the efficiency calculation of the resource holds times of resources for applications. The resource hold time of a resource for an application is the maximum duration of time that the application may lock the resource.

Keywords: Multi-Core, Concurrent Programming, Parallel Programming, inner product.

I. INTRODUCTION

Multi- core processor (MP) is designed to increase efficiency and performance of the system by increasing multi-tasking, parallelism and throughput. A multi-core processor consists of multiple processing units residing in one physical chip having their own set of execution and architectural recourses which differs with the traditional shared memory parallel architectures (SMP) in both hardware and software designs. The numbers of cores in multi-processor are often limited to four or eight whereas in MP designers are thinking to place hundreds or even thousands of cores in a single chip. The cores in MP are more closely coupled than processors in SMPs. In multi core processor system no cache at any level is shared but in MP, L2 and L3 caches are shared by the multiple cores within a chip. Also, the interconnection scheme used for processor-to-processor and processor-to-memory is static for multi-core processor system and dynamic for MP. Similarly, from software design aspect MP also have different challenges than those of SMPs. These include, program or thread scheduling and better load distribution on the available cores. Next is the level of parallelism, MP favors threading level parallelism whereas multi-core processor works better for process or application level parallelism. Some other differences in software design for MP and SMPs may include, design of threads, algorithm decomposition techniques, programming patterns, operating system support etc. The shift towards multi-core processor (MP) from single core processors systems has resulted in several challenges for hardware and software designers. With changes in technology from micrometre to nanometre, there is a significant increase of the number of cores in a

chip. Now, it is computer designer's responsibility to determine a computational structure that can transform the increase in cores into a corresponding increase in computational performance efficiency. This challenge must be dealt with on several fronts, like modification in basic architecture design of each processor (core) to increase single or multi-thread performance, change in the architecture of memory system and development of new programming models for multi-core processors.

The ***SPMD***, (**S**ingle program multiple data), is a new parallel programming model developed as a part of this project. The development of SPMD is motivated with an understanding that existing parallel developments tools do not provide adequate support for general purpose multi-core programming and unable to utilize all available cores efficiently as they are designed for either specific parallel architecture or certain program structure. The SPMD is developed to equip a common programmer with multi-core programming tool for scientific and general purpose computing. The SPMD provides a set of rules for algorithm decomposition and a library of primitives that exploit parallelism and concurrency on multi-core processors. The programming model is serial-like task-oriented and provides thread level parallelism without the programmer requiring a detailed knowledge of platform details and threading mechanisms. It has also many other unique features that distinguish it with all other existing parallel programming models. It supports both data and functional parallel programming. Additionally, it supports nested parallelism, so one can easily build larger parallel components from smaller parallel components. A program written with SPMD may be executed in serial, parallel and concurrent fashion on available cores. Besides, it also provides processor core interaction feature that enables the programmer to assign any task or a number of tasks to any of the cores or set of cores. Besides, the ability to use SPMD on virtually any processor or any operating system with any C compiler makes it very flexible.

II. LITERATURE SURVEY

[1] F. Nemati, T. Nolte, and M. Behnam, "Partitioning real-time systems on multiprocessors with shared resources". 2012. The Author has proposed a heuristic blocking-aware algorithm to allocate a task set on a multi-core platform to reduce the blocking overhead of tasks. Partitioning (allocation tasks on processors) of a task set on a multiprocessor platform is a bin-packing problem which is known to be a NP-hard problem in the strong sense therefore finding an optimal solution in polynomial time is not realistic in general . Heuristic algorithms have been developed to find near-optimal solutions.

[2] M. Behnam, I. Shin, T. Nolte, and M. Nolin. "A synchronization protocol for hierarchical resource sharing in real-time open systems" 2007.The Author has proposed a study of bin-packing algorithms for designing distributed real-time systems was presented. Where in a method partitions a software into modules to be allocated on hardware nodes. In their approach they use two graphs; one graphs which models software modules and another graph that represents the hardware architecture. The authors extend the bin-packing algorithm with heuristics to minimize the number of required bins and the required bandwidth for the communication between nodes. Presented a heuristic algorithm for allocating tasks in multicore- based massively parallel systems. Their algorithm has two rounds, in the first round processes (groups of threads - partitions in this thesis) are assigned to processing nodes, and the second round allocates tasks in a process to the cores of a processor. However, the algorithm does not consider synchronization between tasks.

[3] Rajagopalan, B. T. Lewis, and T. A. Anderson, "Thread scheduling for multi-core platform" 2007.The Author has proposed, a scheduling frame work for multi-core processors was presented by Rajagopalan. The framework tries to balance between the abstraction level of the system and the performance of the underlying hardware. The framework groups dependant tasks, which, for example, share data, to improve the performance. This presents Related Thread ID (RTID) as a mechanism to help the programmers to identify groups of tasks .The grey-box modeling approach for designing real-time embedded systems was presented. In the grey-box task model the focus is on task level abstraction and it targets performance of the processors as well as timing constraints of the system.

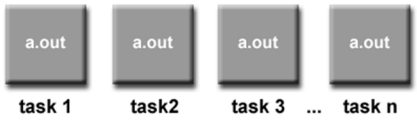
[4] Y. Liu, X. Zhang, H. Li, and D. Qian, "Allocating tasks in multi-core processor based parallel systems" 2007.The Author has proposed a bin-packing partitioning algorithm, the first-fit decreasing (FFD) algorithm for a set of independent sporadic tasks on multiprocessors. The tasks are indexed in non-decreasing order based on their relative deadlines, and the algorithm assigns the tasks to the processors in first-fit order. The tasks on each processor are scheduled under uniprocessor EDF.

[5] K. Lakshmanan, D. de Niz, and R. Rajkumar "Coordinated task scheduling, allocation and synchronization on multiprocessors" 2009.The Author has proposed an investigated and analysed two alternatives of execution control policies (suspend-based and spin-based remote blocking) under MPCP. They have developed a blocking-aware task allocation algorithm, an extension to the best-fit decreasing (BFD) algorithm, and evaluated it under both execution control policies. Their

blocking-aware procedure is great relevance to our propose method, hence we have presented their algorithm in more details in Sections. Together with our procedure also implemented and evaluated their blocking-aware algorithm and compared the performances of both algorithms

III. MOTIVATION

In the past all applications were applied on single core processor with better performance, but also comparison of energy and not suitable scaling. With the recent Era in multi-core helps to increases the scalability and energy management of the processor. This motivates to propose the work in partition which allocates task on multi-core platform in an away that overall amounts of blocking times of tasks are decreased. The parallel task scheduling procedure is built to achieve efficient output and the better resource sharing of the parallel tasks in the multicore processor.



IV. PROBLEM DEFINITION

The single core processor parallel synchronization procedures have been extended to support inters processor synchronization among tasks. However, under task scheduling methods, the single core processor synchronization procedures cannot be reused without modification. Instead, parallel synchronization for task and scheduling procedures have to develop to support resource sharing under task scheduling methods, to get desired efficient output from parallel task scheduling and resource sharing with the minimization of execution time and power consumption.

V. OBJECTIVES

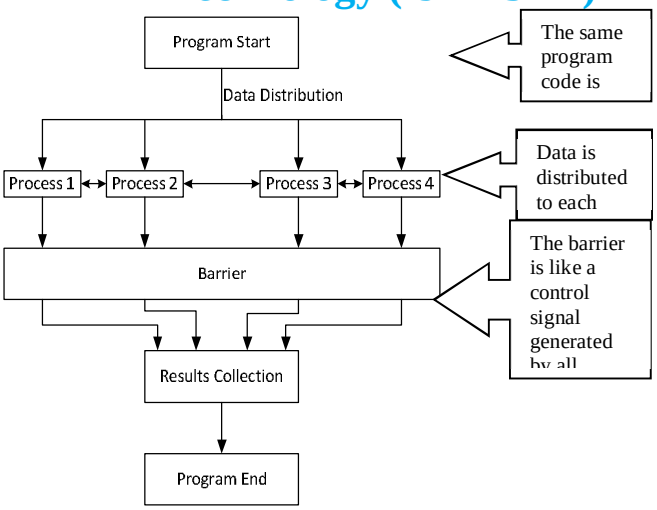
Scalability, Portability and Predictability.
Minimization of the execution time and energy consumption of task.
Increasing the Scalability in multi-core processor.

VI. SYSTEM DESIGNING AND DEVELOPMENT

A. Single Program Multiple Data (SPMD)

SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.
MULTIPLE DATA: All tasks may use different data
SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.SPMD mode is a method of parallel computing, its processors run the same program, but execute different data.
SPMD could get better computing performance through increasing the number of processors/cores

Volume 3 Issue IX, September 2015
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)



B. Advantages of SPMD

1) Locality. Data locality is essential to achieving good performance on large-scale machines, where communication across the network is very expensive.

2) Structured Parallelism. The set of threads is fixed throughout computation. It is easier for compilers to reason about SPMD code, resulting in more efficient program analyses than in other models.

3) Simple Runtime Implementation. SPMD belongs to MIMD, it has a local view of execution and parallelism is exposed directly to the user, compilers and runtime systems require less effort to implement than many other MIMD models.

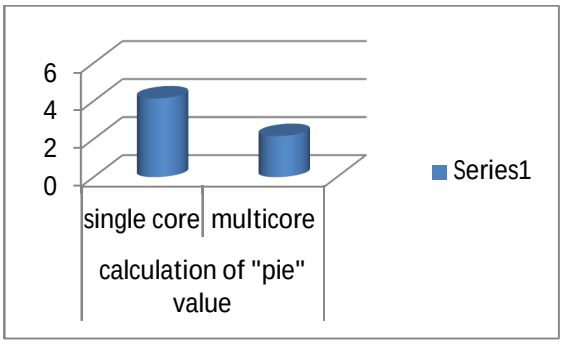
C. Disadvantages of SPMD

SPMD is a flat model, which makes it difficult to write hierarchical code, such as divide-and-conquer algorithms, as well as programs optimized for hierarchical machines.

The second disadvantage may be that it seems hard to get the desired speedup using SPMD.

VII. RESULT AND RESULT ANALYSIS

Obtained result of the program calculating the value of “pi” using both the single and multi-core processor



VIII. CONCLUSION

The results from this project work show that the Multicore programming model provides a simpler, effective and scalable Way to perform any applications on multi-core processors. With the concurrent function of multi-core programming model, the programmer can execute a simple and standard applications algorithm Concurrently on multi-core processors in much less time than that of standard parallel Open-MP. This multi-core programming model will be further worked out for the introduction of some more parallel and concurrent functions and Synchronizing tools.

IX. REFERENCES

[1] F.Nemati and T. Nolte. Resource sharing among prioritized real-time applications on multiprocessors. Technical report April, 2012.
[2] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems. In Proceedings of 7th ACM & IEEE International conference on Embedded software (EMSOFT'07), pages 279–288, 2007.
[3] Rajagopalan, B .T .Lewis, and T .A. Anderson, "Thread scheduling for multicore plarform" 2007.
[4] Y .Liu X . Zhang , H. Li, and D.Qian, "Allocating tasks in multi-core processor based parallel systems" 2007.
[5] K . Lakshmanam, D.de Niz , and R Raj Kumar "Co-ordinated task scheduling, allocation and synchronization on multiprocessor"2009.
[6] F. Nemati, T. Nolte, and M. Behnam. Partitioning real-time systems on multiprocessors with shared resources. In Proceedings of 14th International Conference on Principles of Distributed Systems (OPODIS'10), pages 253–269, 2010.

CASE STUDY: INNER PRODUCT

Testing the applicability of the created ***mclib*** interface to scientific computing problems requires a practical use case scenario. Its solution has to be sufficiently computationally intensive to require parallelization with the need of synchronization at multiple points. This chapter describes such a use case, giving a step-by-step derivation of the computational Inner Product Computation.

A. Algorithm For Inner Product Computation

A simple example of a ***mclib*** algorithm is the following computation of the Inner product α of two vectors

$\mathbf{x} = (x0, \ldots, xn-1)T$ and $\mathbf{y} = (y0, \ldots, yn-1)T$,

$$\alpha = \sum_{i=0}^{n-1} xi\ yi$$

In our terminology, vectors are column vectors; to save space we write them as $\mathbf{x} = (x0, \ldots, xn-1)T$, where the superscript 'T' denotes transposition. The vector $\mathbf{x}$ can also be viewed as an $n \times 1$ matrix. The inner product of $\mathbf{x}$ and $\mathbf{y}$ can concisely be expressed as $\mathbf{x}T\mathbf{y}$. The inner product is computed by the processors $P(0), \ldots, P(p-1)$ of a MC computer with p processors. We assume that the result is needed by all processors, which is usually the case if the inner product computation is part of a larger computation, such as in iterative linear system solvers.

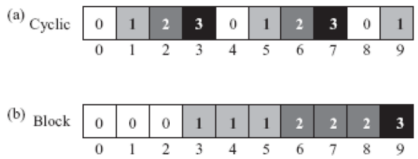


FIG. 1.5. Distribution of a vector of size ten over four processors. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered 0, 1, 2, 3. (a) Cyclic distribution; (b) block distribution.

$$xi \_ \longrightarrow P(i \bmod p), \text{ for } 0 \leq i < n.$$

Here, the mod operator stands for taking the remainder after division by p, that is, computing modulo p. Similarly, the div operator

stands for integer division rounding down. Figure 1.5(a) illustrates the cyclic distribution for $n = 10$ and $p = 4$. The maximum number of components per processor is $\lceil n/p \rceil$, that is, n/p rounded up to the nearest integer value, and the minimum is $n/p = n \text{ div } p$, that is, n/p rounded down. The maximum and the minimum differ at most by one. If p divides n, every processor receives exactly n/p components. Of course, many other data distributions also lead to the best possible load balance. An example is the **block distribution**, defined by the mapping

$$x_i \longrightarrow P(i \text{ div } b), \text{ for } 0 \le i < n,$$

with block size $b = \lceil n/p \rceil$. Figure 1.5(b) illustrates the block distribution for n=10 and p=4. This distribution has the same maximum number of components per processor, but the minimum can take every integer value between zero and the maximum. In Fig. 1.5(b) the minimum is one. The minimum can even be zero: if $n = 9$ and $p = 4$, then the block size is $b = 3$, and the processors receive 3, 3, 3, 0 components, respectively. Since the computation cost is determined by the maximum amount of work, this is just as good as

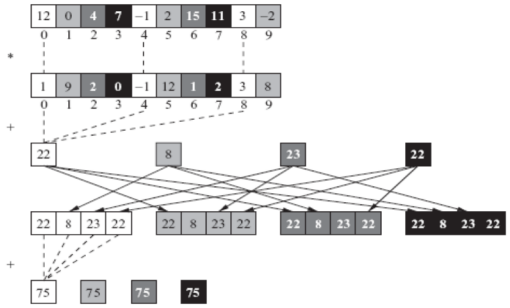


FIG. 1.6. Parallel inner product computation. Two vectors of size ten are distributed by the cyclic distribution over four processors. The processors are shown by greyshades. First, each processor computes its local inner product. For example, processor $P(0)$ computes its local inner product $12 \cdot 1 + (-1) \cdot (-1) + 3 \cdot 3 = 22$. Then the local result is sent to all other processors. Finally, the local inner products are summed redundantly to give the result 75 in every processor.

Inner product algorithm for processor $P(s)$, with $0 \le s < p$.

input: x, y : vector of length n,
distr(x) = distr(y) = ϕ,
with $\phi(i) = i \bmod p$, for $0 \le i < n$.
output: $\alpha = x^T y$.

(0) $\alpha_s := 0$;
for $i := s$ **to** $n-1$ **step** p **do**
$\alpha_s := \alpha_s + x_i y_i$;

(1) **for** $t := 0$ **to** $p-1$ **do**
put α_s in $P(t)$;

(2) $\alpha := 0$;
for $t := 0$ **to** $p-1$ **do**
$\alpha := \alpha + \alpha_t$;

the cyclic distribution, which assigns 3, 2, 2, 2 components. Intuitively, you may object to the idling processor in the block distribution, but the work distribution is still optimal!

Algorithm 1.1 computes an inner product in parallel. It consists of three supersteps, numbered (0), (1), and (2). The synchronizations at the end of the supersteps are not written explicitly. All the processors follow the same program text, but their actual execution paths differ. The path of processor

$P(s)$ depends on the processor identity s, with $0 \le s < p$. This style of programming is called **single program multiple data** (SPMD), and we shall use it throughout the book.

In superstep (0), processor $P(s)$ computes the local partial inner product

$$\alpha_s = \sum_{0 \le i < n,\ i \bmod p = s} x_i y_i,$$

multiplying x_s by y_s, x_{s+p} by y_{s+p}, x_{s+2p} by y_{s+2p}, and so on, and adding the results. The data for this superstep are locally available. Note that we use global indices so that we can refer uniquely to variables without regard to the processors that own them. We access the local components of a vector by stepping through the arrays with a **stride**, or step size, p.

The program text of such an algorithm must contain additional if-statements to distinguish between sends and receives. Careful checking is needed to make sure that pairs match in all possible executions of the program. Even if every send has a matching receive, this does not guarantee correct communication as intended by the algorithm designer. If the send/receive is done by the

handshake (or kissing) protocol, where both participants can only continue their way after the handshake has finished, then it can easily happen that the sends and receives occur in the wrong order. A classic case is when two processors both want to send first and receive afterwards; this situation is called **deadlock**. Problems such as deadlock cannot happen when using puts. In superstep (2), all processors compute the final the cost analysis of the algorithm is as follows. Superstep (0) requires a floating-point multiplication and an addition for each component. Therefore, the cost of (0) is 2_*n/p*_ + *l*. Superstep (1) is a (p − 1)-relation, because each processor sends and receives $p-1$ data. (Communication between a processor and itself is not really communication and hence is not counted in determining *h*.) The cost of (1) is $(p-1)g + l$. The cost of (2) is $p + l$. The total cost of the inner product algorithm is T in prod = 2_*np*_+ $p + (p-1)g + 3l$.

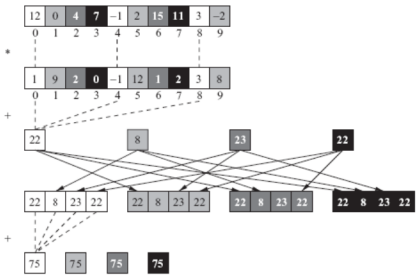


FIG. 1.6. Parallel inner product computation. Two vectors of size ten are distributed by the cyclic distribution over four processors. The processors are shown by greyshades. First, each processor computes its local inner product. For example, processor $P(0)$ computes its local inner product $12 \cdot 1 + (-1) \cdot (-1) + 3 \cdot 3 = 22$. Then the local result is sent to all other processors. Finally, the local inner products are summed redundantly to give the result 75 in every processor.

The relation between Algorithm 1.1 and the function mc inprod.c is as follows. The variables *p, s, t, n, α* of the algorithm correspond to the variables p, s, t, n, alpha of the function. The local inner product α_s of $P(s)$ is denoted by inprod in the program text of $P(s)$, and it is also put into Inprod[s] in all processors. The global index i in the algorithm equals i * p + s, where i I sa local index in the program. The vector component x_i corresponds to the variable x[i] on the processor that owns x_i, that is, on processor $P(i \bmod p)$.The number of local indices on $P(s)$ is nloc(p,s,n). The first $n \bmod p$ processors have _*n/p*_ such indices, while the others have *n/p*. Note the efficient way in which nloc is computed and check that this method is correct,

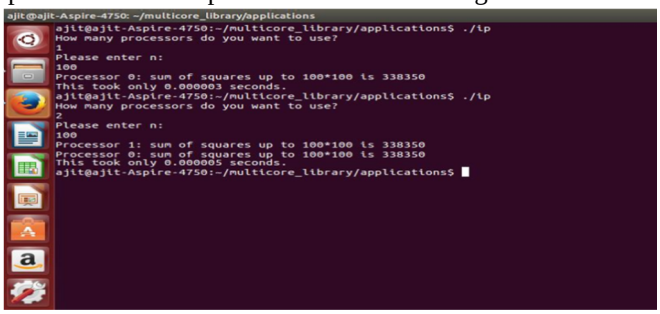
by writing $n = ap + b$ with $0 \le b < p$ and expanding the expression returned by the function.

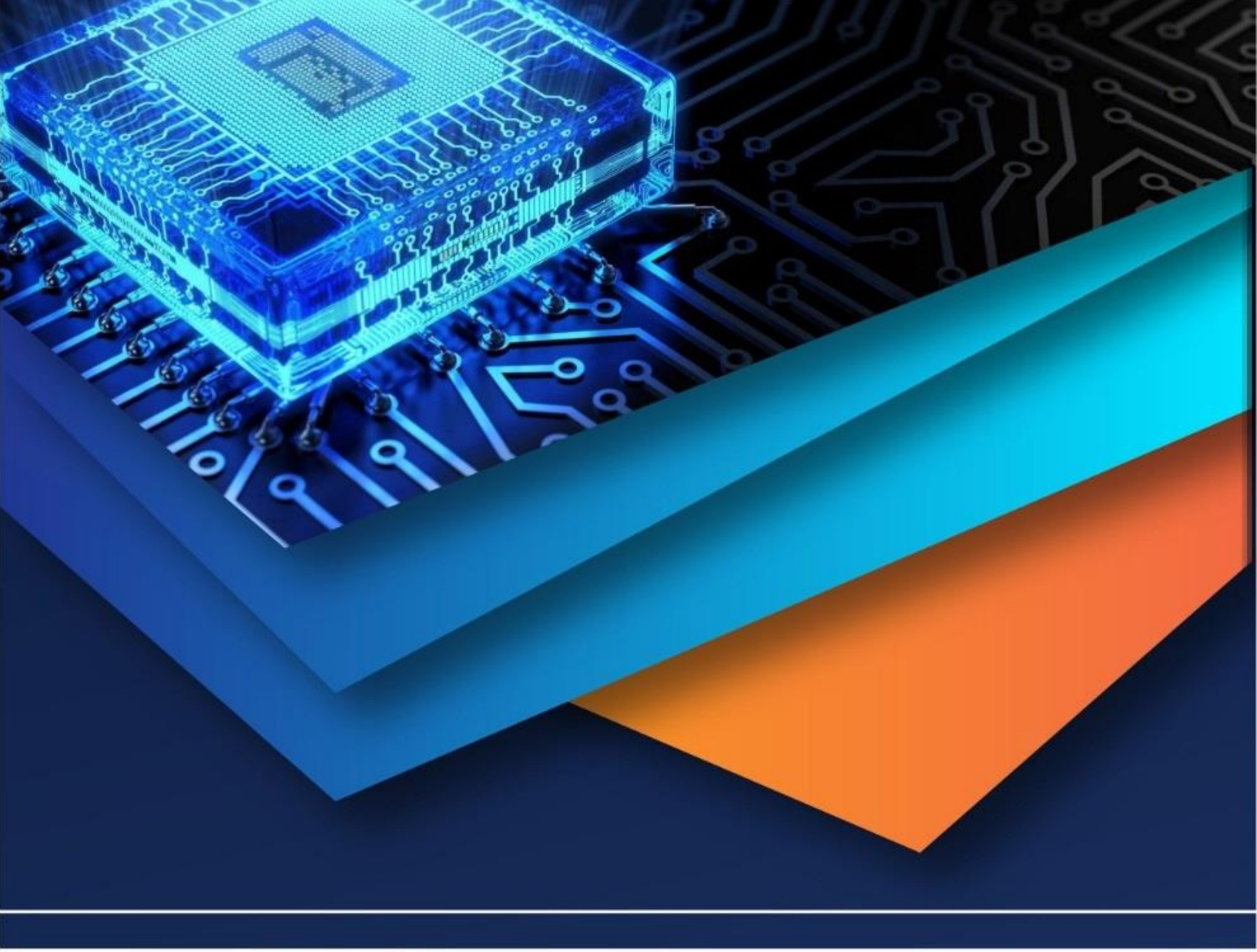
To run the ip program on at most three processors:

```
$ mcrun -npes 3 ip
How many processors do you want to use?
2
Please enter n:
100
Processor 0: sum of squares up to 100*100 is 338350
This took only 0.000003 seconds.
Processor 1: sum of squares up to 100*100 is 338350
```

The program or the user can decide to use less than three processors; here two processors are actually used.

Note that the output of the different processors is multiplexed and hence can be garbled.





10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY