B.Tech. Project Report

on

Study of Parallelization of Travelling Salesman Problem on Shared Memory Multi-Cores

Submitted by

Raghuvar Prajapati (201351003)

Rahul Nalawade (201351017)

Mentors

Prof. Kalyan Sasidhar

Prof. Reshmi Mitra

***Abstract*—Travelling Salesman Problem is an NP hard problem in combinatorial optimization. It has many related applications like Routing problem or DNA sequencing. This work evaluates performance improvement from multiple optimizations on the Travelling Salesman Problem and Dijkstra's algorithm. The paper also consists of the optimization techniques provided by the Charm++ framework. The optimizations used are adaptive grain[1] size control, speculation and prioritized execution, distributed task scheduling, support for branch and bound search. The major results show the performance gain of 30 folds with respect to sequential execution of same algorithms. Overall it has been observed that speculation and prioritized execution has much more significant than the other optimizations.**

***Keywords*—TSP, Dijkstra, Parallel programming, Charm++, Branch and Bound, Simulators.**

I. Introduction

Over the years, multi-core tech have boosted the performance of widely used computing devices and have become an intrinsic part of most of the architectures used today. Such sophisticated increase in the cores of a CPU has gained tremendous popularity. With high performance processors like, Samsung Exynos 8 Octa 8890, Qualcomm Snapdragon 821 MSM8996 Pro, KIRIN 955, TEGRA X1, the necessity of parallelizing overall CPU load is the major hurdle for large class of applications. Future devices will continue this trend of large scale processing, for example Qualcomm has recently announced that their new Snapdragon 820 chipset will come with Adrino 530 GPU which is said to support superior DSLR like photography, and bringing computer vision, virtual reality and photo realistic graphics to your smart phones.

Due to high processing power available in modern computing devices users tend to expect highly interactive applications, which includes web-applications, multimedia, games using high-end graphics. Utilizing the powerful parallel machines to overcome these challenges involves exhaustive state-space search, dynamic scheduling or task prioritization. Moreover, adaptive run-time systems has become an essential part of such parallel programming optimizations.

This paper describes these techniques to boost the performance of multi-core systems. It also summarizes an attempt to simulate travelling salesman problem and Dijkstra problem on various simulators to study it's behavior.

The main objective of the current work is study the programming aspect of NP-hard problem, like TSP or Dijkstra's Shortest path algorithm, and implement efficient techniques, like Adaptive Grain Size Control, Speculation and Prioritized execution, Distributed task scheduling, Support for branch-and-bound search; that increases the yield. In other words, studying different optimization techniques available on various parallel processing simulators, how one can improvise the searching algorithm of finding shortest paths on a given graph.

II. Related Work/Literature Survey

Programs written for different operating systems can be easily analyzed by the simulators. Foregoing studies have shown that there are several types of simulators and frameworks in which the TSP problem can be simulated. Simulations make easy to visualize the application in real time. Much of the ongoing research on simulators in the

[1] grain size of a task is a measure of the amount of work (or computation) which is performed by that task.

parallel framework include GRAPHITE, QEMU, GEM5, SIMCA, and Charm++. These are the simulation environment in which the problem can be programmed and analyzed further. Many programming techniques have been introduced for an application to incorporate these simulators as per their requirement.

In simulation environment, [8] GRAPHITE has been used in open-source, distributed parallel simulator for multi-core architectures. The tool also has been used to give insight of the multi-core processors. It also provides high performance for software development. Many techniques can be adapted using GRAPHITE such as direct implementation, seamless multi-core and multi-node distribution.

The other emulator, [11] QEMU has been used for the full system emulation. It allows OS to run on a virtual machine as well as, in linux user mode emulation, It compiles linux processes for one target CPU which can be implemented on another CPU.[2]

Based on the architectural modelling gem5 is used as a community tool. The main concepts of this simulator are: [6]

- Flexible modeling appealing to a wide scale of users.
- Benevolent to community and high availability
- Rich developer interactions.

SIMCA [3] has been used for the multi-threaded computer architecture. The super-threaded architecture uses the multi-threading concept to execute multiple independent program tasks concurrently. There are some drawbacks of using this simulator like- large amount of process creation and termination of synchronization overhead. SIMCA has been successfully used to simulate several of the SPEC(Standard Performance Evaluation Corporation) benchmark programs [10].

Charm++ has been used an object oriented parallel programming framework language which is based on C++. It gives a clear cut differentiation between the serial and parallel objects used in normal and parallel programmings respectively. Most of the work of the TSP application has been done in this framework because of its suitability and feasible features such as support for dynamic binding, multiple inheritance, overloading and reuse of parallel objects. Charm++ supports unique kind of shared objects 'Chare' which is responsible for message passing, that makes it more efficient and acceptable other than the object oriented languages.

So, Charm++ is an excellent way for the TSP application to apply the concepts and optimization techniques to real-time observations and concurrently benefit the application.

[2]QEMU, a Fast and Portable Dynamic Translator
[3]Simulator for Multi-threaded Computer Architecture

III. The Present Investigation/System Architecture

A. System Architecture

The basic experimental setup in the Charm++ environment is as follows :

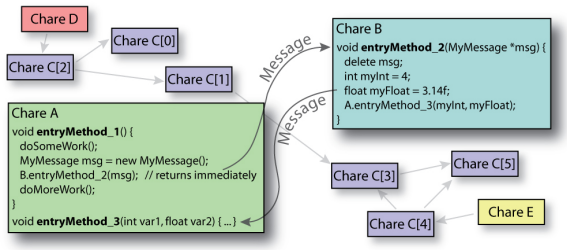

Figure 1: **System's view of a Charm Application**

The above figure represents the general system's view of a Charm application. TSP application has been simulated in the Charm++ framework. From the user's perspective a Charm++ program is simply a collection of Chare objects. [4]. Chare objects can be declared and initialized with the state defined inherently. The Chare objects communicate each other via message passing. When a specific Chare object receives a message, sent by the another Chare object, it will execute an entry method with their message passing component as its argument, to process the message. [2] Charm++ application does not depend on what type of processing element is used, how Chare objects should be assigned to their respective elements.

B. System Specifications

System that has been used to run the application have following specification:

```
Architecture: x86_84
CPU op-mode(s): 64-bit
Byte Order: Little Endian
CPU(s): 4
CPU MHz: 1404.070
L1d cache: 32k
L1i cache: 32k
L2 cache: 256k
L3 cache: 3072k
```

C. The Present Investigation

Several works had already done in the field of object oriented paradigm. Large part of applications from operations research, data mining and even AI could be advantageous from the parallel processing. In many of the available framework Charm++ is such a paradigm which has been used to tackle

[4]Objects created in the Charm++ paradigm is known as **Chare**

the TSP application. The advance and efficient features of the Charm++ framework in parallel programming makes it more adaptive to use for simulating the TSP application.

Charm++ is machine independent an object-oriented asynchronous message passing parallel programming paradigm [12]. Charm++ programs are basically C++ programs. The global variables declarations are hidden in the Charm++ programs. The basic parallel parallel processing components used to improve the parallelization of the programs are also used as an extension in them. Followings are the features and properties of the Charm++ framework that has been encountered while running the TSP application:

1) Structure and Execution Model: Basic Charm++ programs can be seen as the collection of modules defined for the each segment in the Charm++ programs. The modules are defined in a different file. Each modules are defined and declared as:

Messages : Message defined in the Charm++ programs are similar to the structures defined in C++

```
message TypeOfMessage{
 .. // Data members
 }
```

Chare Classes : Chares are the special concurrent objects declared in the Charm++ programs :

```
Chare class ChareName[superclass]{
  ..// Data members
entry:
   EntryPointName(MessageType *MsgPointer){
    // C++ code block
    }
```

Basic Charm++ Calls: There are some macros in the basic Charm++ programs which are similar to the C++. The ***ChareExit()*** calls OS to deliver memory of the running executing Chare object. Terminal input and output from any processor can be defined by the macros as ***CPrintf()*** and ***CScanf()*** calls, which are very similar to C equivalent.

2) Load Balancing and Memory Management Strategies : Depending on the necessity of the application, Charm++ enable programmers to select a strategy from a number of lists of the dynamic load balancing strategies. The selected technique among the dynamic load balancing can be executed on the top of run-time system, based on the requirement of the application. Below are some techniques that has been used in the dynamic load balancing strategies:

1) *Random:* Newly created Chare object has been given to the processors assigned randomly.
2) *Adaptive Contracting Within Neighborhood:* The Chare object that has been newly created, can be balanced within the neighborhood of another Chare objects.
3) *Token:* Tokens can be used as dynamic load balancing concept, which uses prioritized task creation. But this strategy is more knowledgeable, scalable and can be extended to the parallel processing components.

3) Adaptive Grain Size Control: : While designing a parallel application, the parallelization technique is considered significant. One must decompose the application into tasks so as to create enough parallelism, while keeping the overheads of task creation and scheduling to a minimum [7]. These overheads overcome with *Startup and Saturation* techniques. The startup phase begins with enlargement of the root into its children. Or the initial state can be divided into its sub-tasks as a binary search tree. In this stage, task is divided as fine-grained decomposition. On the other hand, in the Saturation phase the purpose is to minimize the amount of overhead sustained in performing the parallel search [13]. The overall technique of decomposing the problem based on adaptive grain size control could be seen as:

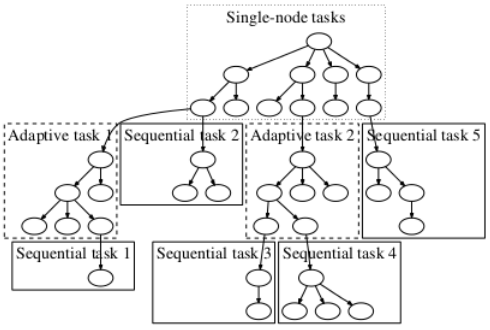

Figure 2: **Adaptive Grain Size Control**

4) Speculation and Prioritized Execution: Working in parallel environment involves some amount of speculative work. Many researchers also discussed that the effect of the speculation leads directly to the speed-ups of the application while increasing the number of processors. The distributed task of the application depends on the prioritize speculative computation performed in it. The value of the priority defined in the binary tree is labeled in the lexicographical order. This scheme have many benefits, like - Reduced Speculation and Reduced memory footprint.

5) Distributed Task Scheduling: To attain a good balance between the loads assigned to the processors, the parallel tasks should be equaly dispersed to the processors. Currently two load balancing strategies have been implemented in this environment.

- *Randomized load balancing strategy* : This technique has been used in assigning tasks to the processor that is created recently. Initially the task is assigned randomly to the processor [7].
- *Randomized work stealing* : It's a popular dynamic load balancing scheme, which is based on the divide-and-conquer technique and can be asymptotically op-

timized. Originally this scheme has shown much improvement in the performance having shared memory systems. In this scheme a newly generated task has been assigned to its local queue [7].

6) Branch and Bound Technique: The branch-and-bound technique is used to prune nodes aggressively and reduce the size of the divided task of an application. The application is set to be work under an already set threshold, beyond that the unnecessary steps would be pruned accordingly.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Traveling salesman problem. Cost of any tour can be written as below :

```
Cost of a tour T = (1/2) * (Sum of cost of
    two edges adjacent to u and in the tour T)
               where u belongs to V

For every vertex u, if we consider two edges
    through it in T, and sum their costs. The
    overall sum for all vertices would be
    twice of cost of tour T.

(Sum of two tour edges adjacent to u) >= (sum
    of minimum weight two edges adjacent to u)

Cost of any tour >= 1/2) * (Sum of cost of
    two minimum weight edges adjacent to u)
               where u belongs to V
```

D. Serial Version of Dijkstra Algorithm

```
dijkstra(int G[][],int n,int startnode)
{
while(count<n-1)
{
 mindistance=INFINITY;
   for(i=0;i<n;i++)
     if(distance[i]<mindistance&&!visited[i])
       {
        mindistance=distance[i];
        nextnode=i;
       }
  //check if a better path exists through
      nextnode
 visited[nextnode]=1;
  for(i=0;i<n;i++)
    if(!visited[i])
      if(mindistance+cost[nextnode][i]<distance[i])
       {
        distance[i]=mindistance+cost[nextnode][i];
        pred[i]=nextnode;
       }
       count++;
   }
}
```

IV. Results and Discussions

A. Contributions

Raghuvar: Here, I would like to summarize my individual contribution to the project. I have spent considerable amount of time working on specific tasks of the project, aligning to my area of expertise. The key components on which I have worked on are summarized as:

1) Explored different types of simulators available for the parallel programming framework specifically worked on gem5 simulator and Charm++.
2) Studied or gone through at most every available documents about the Charm++ parallel programming framework used for the TSP application.
3) Implemented a Shortest path algorithm “Dijkstra’s Algoriithm ”in Charm++ framework so that it could be generalized for almost every such application by using Charm++ framework.
4) Studied and analyzed the different parallel programming features such as speed-up, performance, types of parallelism and many more.

Rahul: Working for finding best performance analyzing tool that could show the direct application of parallel programming concepts, I can briefly summarize my contribution to our B.Tech. project:

1) Designed the problem flow-chart by creating required input to the implementable TSP code. It consists of making a random plot of N nodes with connecting them by K(k=3) Nearest Neighbors so as to create input adjacency matrix as simple as possible.
2) Explored different types of simulators (Graphite, Simca, SimpleScaler and Hase) available for the parallel programming framework.
3) Implemented TSP algorithm with Adaptive Grain-size Control, Speculation and Prioritization execution, Distributed task scheduling and Support for branch-and-bound search.
4) Worked on Charm++ benchmarks and studying various models and techniques.

B. Simulation Results

Both the Dijkstra’s algorithm and TSP shows substantial increase in the execution time on scaling the size of the problems. From fig 3, the TSP execution time is scaling up exponentially and Dijkstra’s execution time seems to be increasing but not with the same factor. This difference can be attributed to the fact that, the TSP models every permutation with fact(N) ways whereas Dijkstra’s algorithm has complexity of square(N).

Below figure #4 represents the results of how execution time varies with the number of threads. The figure has been sketched by using the reduced Over-Decomposition concept of Charm++. Over-decomposition is a process which breaks the application into data and work units excessively. As we can

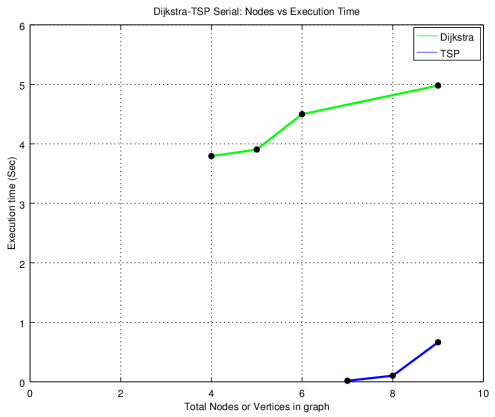

Figure 3: **Dijkstra-TSP Serial: Nodes vs Execution Time**

clearly see that the Execution time keeps on going down while increasing the number of threads.

The statistical data about parallel implementation of the Dijkstra's algorithm is shown in table mentioned below:

Number of Vertices	Number of Threads	Execution Time	Chunk Size
4	2	0.665000s	2
4	4	0.405000s	1
8	2	2.862000s	4
8	4	1.741000s	2
8	8	1.402000s	1
12	2	5.846000s	6
12	3	4.364000s	4
12	4	2.953000s	3
12	6	1.625000s	2
12	12	0.603000s	1

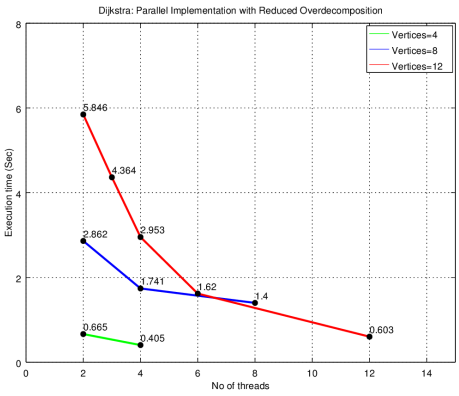

Figure 4: **Dijkstra: Parallel Implementation with Reduced Over-decomposition**

Below figure #5 represents the results of how Chunk[5] Size varies with the execution time for different set of problems means increasing the problem size. The figure uses the concept of adaptive grain size control techniques of Charm++. In the application chunk size has been calculated as number of vertices/number of threads. As the chunk size kept on increasing the Execution time changes accordingly.

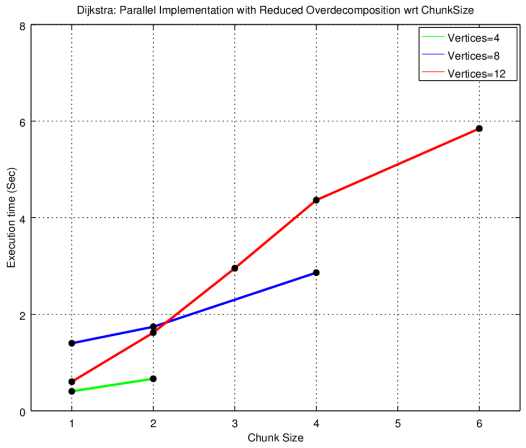

Figure 5: **Dijkstra: Parallel Implementation with Reduced Over-decomposition w.r.t Chunk Size**

For TSP, the graph with 7 nodes simulated over multiple threads shows optimal performance for threads more than 4 and less than 16. With numbers of processors 4, Fig 6 shows the simple parallel implementation without any supporting techniques.

Conventions used for naming Graphs:
For an image named as TSP-abc-d ...
If *a=1*, implementation is done in parallel. Else *a=0*.
Similarly, *b=1* refers to implementation done with Speculation and Task Prioritization. Else *b=0*.
c=1 indicates implementation is done with Adaptive Grain Size Control.
d represents graph size (total no vertices).

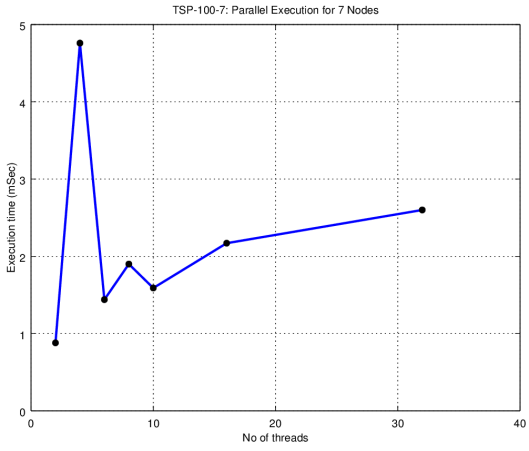

Figure 6: **TSP-100-7: Parallel Execution for 7 Nodes**

With fewer grain size at initial execution, the number of subtasks (grains) are gradually increased. This adaptive grain

[5]A discrete fragment of work load or data structure obtained after partitioning complex programs

size control technique is used in Charm++ and is applied to the algorithm proposed above. Fig 7 shows the reduction in execution time w.r.t. the previous parallel implementation with different thread sizes.

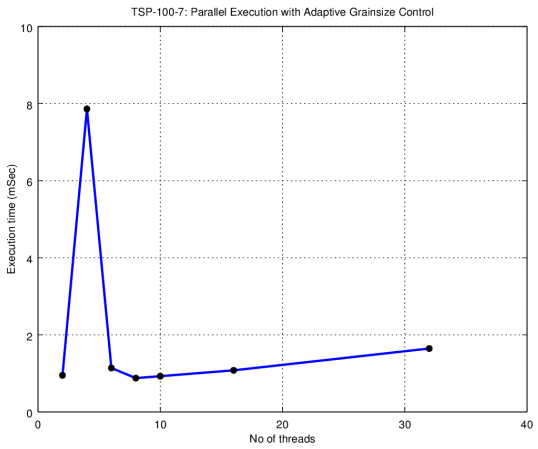

Figure 7: **TSP-101-7: Parallel Execution with adaptive grain size control**

Speculation and task prioritization techniques are used to predict the permutation which is able to give a solution or else the permutation is skipped. Thus, the cost of optimization can be heavily reduced and Fig 8 shows substantial speedup as the performance is gained over multiple threads.

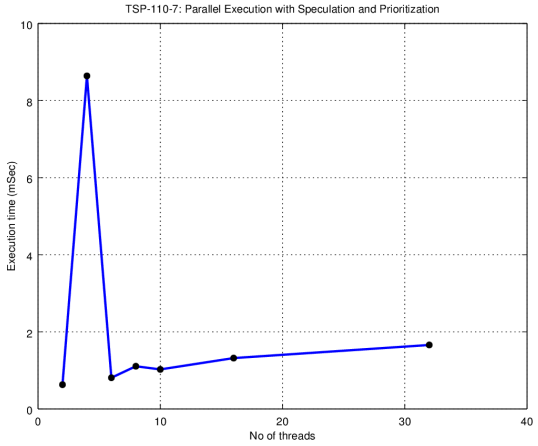

Figure 8: **TSP-110-7: Parallel Execution with Speculation and Prioritization**

By implementing both the above techniques, Adaptive Grain Size Control and Speculation and prioritization of tasks, the optimal performance is gained as memory footprint is saved as well as unnecessary computations are skipped in the implementation. Fig 9 shows the comparative time for most efficient algorithm for 7 nodes.

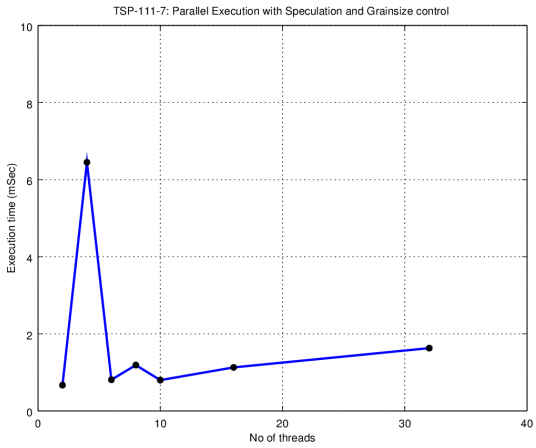

Figure 9: **TSP-111-7: Parallel Execution with Speculation and Grain size control**

By implementing the proposed algorithm for 8 and 9 nodes, with 4 threads, the comparative execution times can be used in determining the significance of each parallel programming technique. From fig 10, one can easily infer that the Speculation and Prioritization of tasks provides much significant gain compared to Adaptive Grain Size Control.

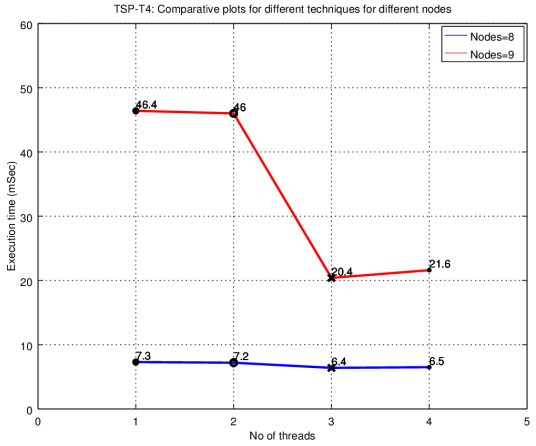

Figure 10: **TSP-4threads: Comparative plots for different techniques for different nodes**

C. Projections Tools Results

Projections is a visualization tool to help you understand and analyze what it happening in your parallel (Charm++) program. To use Projections, Charm++ is compiled with tracing enabled and program is compiled with the Projections trace-mode.

Following images show some statistics about the application after running it into the Charm++ framework. The first image show the profile of usage for all the processor which was initialized at run time.

The second one shows the communication between the Chare objects and the processors. Messages communicated over the network has been shown via diagram.

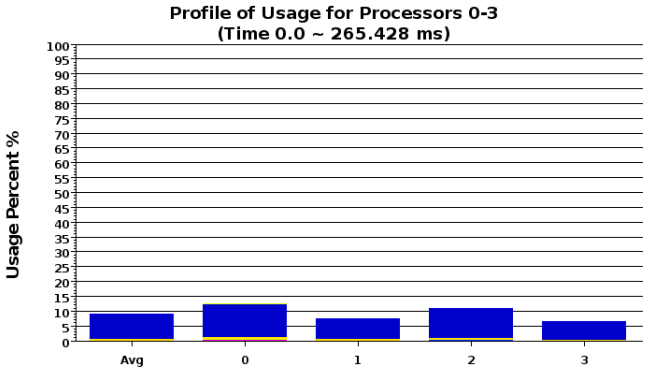

Figure 11: **Usage Profile View of the application**

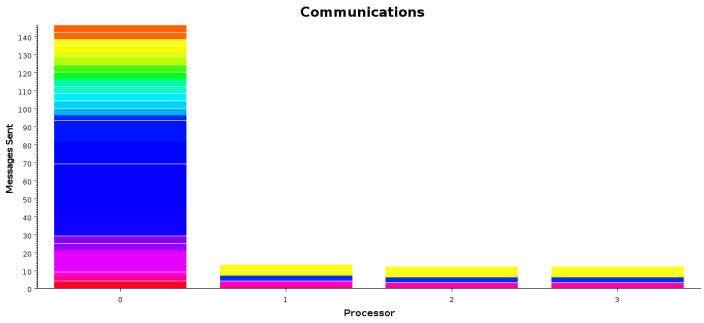

Figure 12: **Projection plot utility view of the application**

V. Conclusions and Future Work

Dijkstra's algorithm and Travelling Salesman Problem were implemented using different optimization techniques. Both the problems required exhaustive search on their large state spaces. Using these problems to implement parallel programming optimizations to analyze the performance gain with different techniques proved relevant. Dijkstra's algorithm is implemented with reduced over-decomposition, where as TSP is implemented using adaptive grain size control, speculation and prioritization execution, and branch and bound search. Besides, trying to work with different simulators like Charm++ proved to be helpful in understanding the system architecture, it's parameters, message passing, caches, and speedups. Implementing both Dijkstra's algorithm and TSP, the performance gain of 10 folds and 30 folds is attained with respect to their sequential counterparts respectively.

VI. Acknowledgement

Apart from our efforts, there are many peoples who supported us directly and indirectly with best of their efforts. We take this chance to express our special thanks to the people who have been contributory in accomplishment of this project.

We would like to show our greatest acknowledgment to Prof. Kalyan Sasidhar and Prof Reshmi Mitra. We are very thankful for their support and help. We would also like to thank Charm++ contributors who helped us with best of their ability to solve our doubts during the projects. Without their assistance, the concepts used in this project would not have been compiled. Also, we would like to appreciate people whose guidance in any form proved beneficial.

References

[1] L. V. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, *et al.*, "Charm++ for productivity and performance-a submission to the 2011 hpc class ii challenge, parallel programming laboratory, department of computer science, university of illinois, 2011. friston, kj (2011). functional and effective connectivity: a review," *Brain Connectivity*, vol. 1, no. 1, pp. 13–36.

[2] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, *et al.*, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 647–658, IEEE Press, 2014.

[3] G. Zheng, L. Shi, and L. V. Kalé, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Cluster Computing, 2004 IEEE International Conference on*, pp. 93–103, IEEE, 2004.

[4] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *ACM Sigplan Notices*, vol. 28, pp. 91–108, ACM, 1993.

[5] L. Kale, "A tutorial introduction to charm," *Parallel Programming Laboratory Internal report*, 1992.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[7] Y. Sun, G. Zheng, P. Jetley, and L. V. Kalé, "Parssse: An adaptive parallel state space search engine," *Parallel Processing Letters*, vol. 21, no. 03, pp. 319–338, 2011.

[8] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, IEEE, 2010.

[9] L. V. Kale and G. Zheng, "Charm++ and ampi: Adaptive runtime strategies via migratable objects," *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pp. 265–282, 2009.

[10] J. Huang and D. J. Lilja, "An efficient strategy for developing a simulator for a novel concurrent multithreaded processor architecture," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 1998. Proceedings. Sixth International Symposium on*, pp. 185–191, IEEE, 1998.

[11] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[12] L. V. Kale, "Introduction to charm++ concepts," 2011.

[13] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale, "An adaptive framework for large-scale state space search," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1798–1805, IEEE, 2011.