Architecture Aware Assembly Like Parallel Constructs: Towards Power and Performance Efficient Heterogeneous Cores

A Thesis Proposal
Submitted
to
WARan Research FoundaTion
(WARFT)

by
Prashanth Thinakaran
Senior Research Trainee

In Partial Fulfillment
of the Requirements for the
Research Awareness Program and Training (RAPT)
at
WARFT

Vishwakarma - High Performance Computing Group

WARan Research FoundaTion (WARFT)
Chennai, India
URL: http://www.warftindia.org
email: waran@warftindia.org
Phone: +91-9840218651

January 2013
Architecture Aware Assembly Like Parallel Constructs: Towards Power and Performance Efficient Heterogeneous Cores

Approved by:

Prof N.Venkateswaran, Advisor
Founder-Director, WARFT

Date Approved: ____________________
ABSTRACT

Supercomputing clusters based on thousands of heterogeneous cores are gaining momentum in the multi-core era because of its potential to cater the needs of several classes of applications and also for its energy and compute efficiency. The Exa-scale computing demands the research thrust on programming models models to achieve exa-scale performance. This model should be capable of bridging the gap between the architecture components and programming models, in such a way leading to projected exa-scale peak performance greatly reducing the power requirements. This thesis focuses on evolving power and performance efficient parallel programming model called WarftPPL and associated constructs that handles the complexity at the cluster level, best suited for grand challenge applications. The framework is also intended to reduce the onus laid on the compiler to generate generic instructions to cater to the thousands of CPU and GPU cores. Further locality aware and communication efficient constructs are needed to reduce the energy consumed through data movement across cores in a cluster environment. The current day parallel programming models neither address these issues nor have parallel constructs focusing on power issues. Recent works on parallel programming models concentrates more on the cosmetic user-friendly higher level constructs which often hurts the productivity of compiler by adding more code framework to sort through in analyzing the computation thus lays the burden on the compilers. Further the complexity is increased many folds by optimizing the indirect indices and unraveling method calls, which are the semantics to be bare by the programmer. Hence there is a immediate need for a PPL that is capable of achieving high performance by harnessing the heterogeneity in the architecture and also consume low power by adopting novel techniques for programming constructs and compiler optimizations. Initial pilot simualtions of WarftPPL over other HLLs marks speedups in compilation, execution and runtimes.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................ iii
LIST OF TABLES ................................................................................................ vi
LIST OF FIGURES ............................................................................................... vii
I ORIGIN AND HISTORY ................................................................................. 1
  1.1 Modern Parallel Programming Languages ........................................... 1
    1.1.1 Partitioned Global Address Space (PGAS) .................................... 1
  1.2 Challenges Particular to Exascale .......................................................... 5
    1.2.1 Parallelism ...................................................................................... 5
    1.2.2 Heterogeneity ................................................................................ 6
    1.2.3 Hierarchy ....................................................................................... 6
    1.2.4 Communication and Synchronization ........................................... 6
    1.2.5 Energy ............................................................................................ 7
  1.3 WARFT initiative towards low power supercomputing ..................... 7
    1.3.1 Proposed CUBEMACH Node Architecture: ................................. 7
    1.3.2 Design Features of the novel Cluster architecture ..................... 10
    1.3.3 WIMACS: Warft India Many Core Simulator .............................. 15
II PROPOSED RESEARCH: WPPL & CONSTRUCTS ................................... 16
  2.1 Recent trends in parallel languages ....................................................... 16
  2.2 Towards PPL catering heterogeneous paradigm ................................ 17
  2.3 Bridging the gap between the application and architecture ............. 18
  2.4 Algorithm based parallel execution model ........................................... 19
  2.5 Integrated Constructs for Computation, Communication & Parallelism 20
  2.6 WPPL Execution Flow ........................................................................ 21
III WORK COMPLETED ................................................................................... 25
  3.1 PPL Model Evaluation ........................................................................ 25
  3.2 Execution Time .................................................................................... 25
3.3 Cache access .................................................. 25
3.4 WPPL Code Snippets ........................................... 27

IV WORK TO BE COMPLETED ........................................ 28
4.1 WPPL Compiler ................................................... 28
4.2 Network Processor constructs ................................. 28
4.3 WPPL based Application Execution in WIMACS ........... 28
4.4 Porting Scientific Libraries .................................... 29

REFERENCES ......................................................... 30
LIST OF TABLES

2.1 Looping and Serialization Constructs . . . . . . . . . . . . . . . . . . 22
2.2 Model, Mode, Atomic Constructs . . . . . . . . . . . . . . . . . . . . 23
2.3 Communication and Synchronization Constructs . . . . . . . . . . . . 23
2.4 Control and Exception Constructs . . . . . . . . . . . . . . . . . . . . 24
LIST OF FIGURES

1.1 Top500 Exa-scale Performance predictions ......................... 2
1.2 Energy spent on Data Movement ............................................. 4
1.3 Exa-scale Predictions Report .................................................. 4
1.4 Generic ALFU Architecture ..................................................... 9
1.5 Hierarchical Compiler On Silicon ........................................... 10
1.6 On-Core Network Interconnections ......................................... 11
1.7 CUBEMACH Design paradigm ............................................... 12
1.8 Simultaneous Execution of Multiple Application ....................... 13
1.9 SMAPP Plane ................................................................. 13
1.10 WARFT Framework for Future Generation Clusters .................. 15
2.1 FLOP/sec & Data movement ................................................. 16
2.2 Warft Parallel Programming Language Execution Flowchart ........... 21
3.1 Execution time comparison ................................................ 26
3.2 Cache Access Comparison .................................................... 26
3.3 WPPL Snippet Codes for Different Classes of Algorithms ............. 27
CHAPTER I

ORIGIN AND HISTORY

1.1 Modern Parallel Programming Languages

Recent research focuses in the development of parallel programming models which supports a multithreaded execution and striving to achieve high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. The models have the provision of data placement and tasks on a target architecture in order to tune for locality. Few of the models also supports supports global-view data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. In contrast to many previous higher-level parallel languages, PPL are designed around a multiresolution philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require. Chapel supports code reuse and rapid prototyping via object-oriented design, type inference, and features for generic programming. While most of the PPL are built on the concepts and syntax from many previous languages, their parallel features are most directly influenced by ZPL, High-Performance Fortran (HPF) which are greatly inspired from the languages C and Fortran.

1.1.1 Partitioned Global Address Space (PGAS)

Unified Parallel C (UPC) [5] evolved based on several extensions to ISOC99 is a popular parallel programming language in recent times. UPC is known for its Partitioned Global Address Space (PGAS) [6] feature. There are several flavors of UPC available that comes with variety of associated compilers and runtimes compatible to commodity clusters like Blue Gene, Cray, etc.. The Berkeley’s implementation has
support for high-performance interconnects like Infiniband through portable runtimes [5].

Since UPC is a recently developed language, there are not many real-life scientific libraries and applications using it as their primary parallel programming model. It requires a lot of investment by domain scientists for porting their libraries and simulation models built on UPC and at the same time the portability should also be future proof at least for the next 10 years owing to the investment made in porting the applications. Hence a futuristic approach must be adopted in developing scientific applications and algorithmic libraries [16] [17]. It is well known fact that MPI is used to parallelize several legacy applications and libraries. Hence there are several approaches adopted to establish a unified parallel environment. Several PGAS implementations which are as follows:

- Library -based Global arrays, open Shared memory [9]
- Compiler-based UPC [5], Co-array Fortran
- Language-based x10, chapel, Fortress[10] [13]

Figure 1.1: Top500 Exa-scale Performance predictions
The Applications types whose data access patterns are regular could be programmed in MPI [12], thus could have the data items partitioned across nodes. The programming model evolved must be robust in supporting asynchronous data movement, while the lions share of scientific libraries being irregular applications, in which case the logically shared memory is helpful to a larger extent. Thus developing a hybrid programming model catering the needs of scientific community and also architecture aware is inevitable.

The major challenges for software and library design are:

- Synchronization
- Communication
- Mixed-precision methods
- Auto tuning-hardware independent algorithms
- Fault-resilient algorithm models

Also the recent trend in computing systems, clearly shows that the performance demands to achieve Exa-scale computing could be satisfied only by the year of 2020 and it is also important to be noted that it should be achievable given the power constraint of 20MW consumption for the developed Exa-scale machine [7] [15].

In multicore environment considerable amount of energy is wasted in data movement across cores and chips. The figure depicts the energy spent on core-to-core, core-to-chip, memory interaction [4]. The Figure[1.3] shows the system memory, node concurrency and total concurrency of threads/process/warps (based upon the level of abstraction) is expected to multiply by 1000 folds in ten years. It is been clearly projected that the node memory bandwidth is expected to increase by hundred
to thousand folds. Hence the table clearly shows a research thrust in the areas of concurrency and inter-node bandwidth [7].

Figure 1.3: Exa-scale Predictions Report

Thus there is a very strong need for locality aware and communication efficient parallel programming paradigm to restrict the data movement directly reducing the power consumption and thereby improving the application productivity. It is also to be noted that the proposed programming model should be capable of achieving the
shared memory abstraction with good data partitioning across nodes by minimal the
data movement across cores/nodes.

1.2 Challenges Particular to Exascale

The future generation high performance systems would have thousands of heterogeneous cores. Therefore it is expected that the system concurrency would be around hundreds of millions which would become tremendous challenge for system software to manage and for applications to get good performance at this level of parallelism. Almost all of todays large-scale applications use the message-passing programming model (MPI) together with traditional sequential languages (C, Fortran, C++), but new architectures with many cores per chip and parallelism in the millions are expected to make this programming model more problematic and less productive in the future. Thus new approaches are needed. In order to facilitate the utilization of the extreme scale resources, new programming models and languages must be explored.

1.2.1 Parallelism

Hybrid approaches such as MPI+OpenMP or even MPI+OpenMP+CUDA are being used to express multiple levels of parallelism in application programs. As codes become more irregular and dynamic, it becomes more important to support a programming layer where parallelism is dynamic, with mapping possibly controlled at a lower layer. Therefore deciding a right mix of details specified by the programmer, which is effectively derived by the compiler to generate efficient parallel codes suitable to the architecture. Thus the programmer is expected to know every aspects of his target architecture, thus developing programs that bonds strongly with the architectural resources.
1.2.2 Heterogeneity

Exascale computers will likely combine conventional cores with GPU-like SIMD cores and, possibly, cores that support high levels of concurrent multithreading. The different cores present different programming models at the bottom of the stack. Different exascale systems are likely to have a different mix of such engines, with differing architectures, and differing connectivity. Thus defining a level of abstraction with in the comfort limits of the programmer becomes inevitable.

1.2.3 Hierarchy

Many codes are currently written with a one-level hierarchy (all MPI codes), with multiple MPI processes per node; an increasing number of codes are written with a two-level hierarchy (MPI+OpenMP). Exascale systems are likely to exhibit deeper hierarchies. Current approaches to handle the HW hierarchy expose the size of each level of the system: e.g., number of cores per node and number of nodes. This impairs code portability. One alternative approach, which has been explored by the Cilk project, is to express parallelism using recursive divide-and-conquer constructs; these can be naturally mapped onto any hierarchical hardware (similar to cache-oblivious algorithms that map to any storage hierarchy) but, as currently implemented, may not preserve locality.

1.2.4 Communication and Synchronization

The communication and synchronization mechanisms used intra-node are very different than those used inter-node not only in performance, but also in semantics. This difference is currently exposed in the programming layers used by application programmer. Exascale systems may add one more layer of complexity; e.g., in a chip where cores are partitioned into clusters, with coherent shared memory inside cluster and non-coherent shared memory across clusters.
1.2.5 Energy

Energy is becoming an essential resource in the exascale regime. Energy consumption is largely controlled by reducing the movement of data, be that access to main memory or interprocessor communication. In addition, future systems will provide increasingly fine-grain mechanisms for controlling the energy consumption of various subsystems. One needs to decide at what level those mechanisms become visible to the software and to the programmer.

1.3 WARFT initiative towards low power supercomputing

Waran Research Foundation has contributed towards low power and high performance node and cluster architecture over the past ten years. One of the major initiatives is the Custom Built hEterogenous Multi-core ArCHitectures CUBEMACH [1] [2] [3] [4] [19]. The features of the CUBEMACH node architecture are briefed below and the architecture is pictorially depicted in the Figure[1.6].

1.3.1 Proposed CUBEMACH Node Architecture:

Power and performance have been pre-eminent design constraints for conventional processors. But, with increasing computational demand and node architectures moving towards SuperComputer-On-Chips (SCOCs), computational efficiency is being emphasized over peak performance. Therefore, suitability of the architecture to the application must also become a pre-eminent design constraint for future HPC systems. A shift in design paradigm is considered to be a "stake through the heart" for application developers. But developers have shown tendencies to move to better paradigms from FORTRAN to C++ and parallel environments, provided they are enthused. A CUsutom Built hEterogeneous Multi-core ArCHitecture(CUBEMACH) is proposed that offers a possibility of increased resource utilization, which is exploited by running traces of multiple applications in the same node, without space or time...
sharing, thus enabling the node to cater to a wider class of applications is proposed.

1.3.1.1 Algorithm Level Functional Unit (ALFU) Accelerating the future generation scientific libraries

Higher order functional units, for their ability to compute an entire algorithm of reasonable input size are also referred to as Algorithm Level Functional Units (ALFUs). Different groups of diversified coarse grain functional units, form the heterogeneous multi-functional cores or SCOC IP cores. The main advantages being reduced control complexity, cache misses and memory accesses [3].

1.3.1.2 Algorithm Level Instruction Set Architecture (ALISA)

The efficiency of node architecture must be extracted by designing an appropriate Backbone Instruction Set Architecture (ALISA) that will reflect the workload at a higher level. A single ALISA instruction encompasses the data dependencies associated with several equivalent ALU instructions and helps in minimizing the number of cache misses. ALISA instructions operate on a higher level and require less fetch from memory and therefore the power dissipation will drastically reduce even while triggering many functional units simultaneously. In addition, the control dependencies (context switches) involved for executing a large problem is lesser in ALISA than in conventional ISA, which again leads to reduced cache misses. Figure[1.4] shows that the instructions of multiple application can co-exist in the same pipeline of a single ALFU.

1.3.1.3 Compiler On Silicon Novel compilation and scheduling paradigm

The Compiler-On-Silicon [2] is a hierarchical hardware based compiler cum scheduler. A Primary Compiler On Silicon (PCOS), the first stage of hierarchy, acts as an input interface to the node to obtain the applications which are in the form of the higher language libraries from the overlying host system. The libraries are received as packets of instructions and are broken down into sub-libraries at the PCOS and
are scheduled to the second stage of hierarchy The Secondary Compiler On Silicon (SCOS). SCOS assembles the sub-libraries, analyzes the dependencies and issues ALISA Control words to trigger many HLFUs in parallel. The SCOS Scheduler plays the most prominent role in the Complier On Silicon architecture, responsible for scheduling ALISA control words to various underlying ALFUs and Scalars.

1.3.1.4 On Code Network (OCN) Architecture serving the data hungry heterogeneous cores

Considering the communication complexity of very large scale heterogeneous multi-cores, the architecture of the NoC should be hierarchical and scalable. Conventional grid-based Network-on-Chip (NoCs) cannot suffice the above mentioned highly critical bandwidth requirements, thereby necessitating the need for a high bandwidth cost-effective NoC for large scale heterogeneous multi-core architectures. CUBEMACH design paradigm uses a novel cost effective On Chip Network called the On Core Network Architecture (OCN). The hierarchy of OCN is emphasized by the presence of a Sub-Local Router for a group of ALFUs (population), a Local router for
across population communication. While populations of ALFUs form a core, global routers are used to establish communication across them. The cores and the units connected to OCN are H-Tree structure based to avoid skew in clock and communicational latency. Thus it overcome the shortcomings of the conventional NoCs and its unsuitability with regard to Super Computer On-Chip, the CUBEMACH architecture uses a novel cost effective on chip Network (OCN)

1.3.2 Design Features of the novel Cluster architecture

Making better design choices would improve the execution time of large scale applications, which are currently predicted to be in Exa-op in few years. Future supercomputing models should also address critical design aspects like reliability, fault tolerance and low power issues which are increasingly becoming important design criterions. This section discusses the scope for improvement in the design features of current generation clusters in order to meet the requirements of performance greedy
hybrid applications, also taking into consideration the operating cost factor.

1.3.2.1 **Cluster level Multiple Application Execution Model: without space time sharing**

Novel cluster paradigm and silicon operating system which is a competent cluster design revolves around an execution model to aid the execution of multiple independent applications simultaneously on the cluster SMAPP [4], leading to cost sharing across applications. The execution model should envisage simultaneous execution of multiple applications (running traces of multiple independent applications in the same node) and on all the partitions (nodes) of a single cluster, without sacrificing the performance of individual application, unlike in the current cluster models. Performance scalability is achieved as we increase the number of nodes, the problem size of the individual independent applications, due to non-dependency across applications and hence increase in the number of non-dependent operations as the problem sizes of the applications get increased and this leads to better utilization of the unused resources within the node.

![Figure 1.6: On-Core Network Interconnections](image)
Figure 1.7: CUBEMACH Design paradigm
Figure 1.8: Simultaneous Execution of Multiple Application

Figure 1.9: SMAPP Plane
1.3.2.2 *Silicon Operating System: Towards low power clusters*

In the current execution model, workload of a single application is mapped on to a set of nodes, which does the work of load balancing across the node of a cluster. The node is usually empowered with a stripped kernel, which performs the core OS functionalities such as memory management, process scheduling, I/O handling and interrupt handling. But in the context of the proposed execution model, a new OS paradigm is required for handling the complexities associated with parallel mapping and data tracking of the huge amount of data associated with the different applications. In this scenario, the reliability of the operating system is of paramount importance as the integrity of I/O data sequencing is critical, particularly when dealing with million node clusters. Thus the capability of the cluster to stomach the complexities involved in multiple application execution lies in an efficient OS design.

A software OS may not be proficient enough to exploit the power of the underlying MIP based nodes as well as to meet the computational speeds of such nodes. In view, we have resorted to a hardware based operating system termed as SILICOS, the functionality of which is distributed across the primary and the secondary host planes by designing suitable architectures for the primary and secondary host plane processors to incorporate the functionalities of the cluster operating system. Assembly level parallel programming model leads to improved performance and reduced power in clusters.

Figure[1.10] captures the entire WARFT framework towards low power and high performance computing. Where in Silicon Operating System (SILICOS), The Compiler on Silicon (COS), Algorithm Level Functional Units (ALFU) in heterogeneous many cores and Simultaneous Execution of Multiple applications (SMAPP), Simultaneous Multiple Algorithm Execution (SMAG) in contrast to Simultaneous
Multi-threading (SMT), all together constitute towards drastic power reduction due to the replacement of software and associated space time overhead by hardware means.

1.3.3 WIMACS: Warft India Many Core Simulator

As a milestone step towards automating the design of supercomputing clusters, a heterogeneous multi-core processor simulator that encompasses the various parameters involved in CUBEMACH design paradigm has been developed. WIMAC simulator [18] has been developed is to our best knowledge, the first ever clock driven, cycle accurate heterogeneous multi-core processor simulator. It captures all the parameters that encompasses the node architecture in a detailed fashion and Further details about WIMAC simulator and its features is available in this WARFT’s website.
CHAPTER II

PROPOSED RESEARCH: WPPL & CONSTRUCTS

2.1 Recent trends in parallel languages

The current Parallel Programming Languages are categorized into data parallel languages, explicit communication model and functional languages. These parallel languages stress either only on data parallelism as in C or FORTRAN or only on communication as in NESL [8]. No single language has been developed which can handle both data parallelism and communication model efficiently focusing the futuristic architectural demands.

Major challenge in devising an algorithm is that, it should be capable of harnessing the utmost parallelism from the underlying hardware and also should be future-proof, scalable to thousands of cores effectively. Future generation application libraries should be carefully programmed, keeping the expense of data movement in mind. The current trend shows that the energy required for data movement is much more costly than compared to a logical operation [7] [15].

<table>
<thead>
<tr>
<th>2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP FMADD flop</td>
</tr>
<tr>
<td>DP DRAM read</td>
</tr>
<tr>
<td>Local Interconnect</td>
</tr>
<tr>
<td>Cross System</td>
</tr>
</tbody>
</table>

**Figure 2.1:** FLOP/sec & Data movement

With increasing complexity of the application, the programming model needs
to be simple and expressible and also allow programmers to represent complex logic efficiently. For this, the Parallel Programming Language (PPL) model should be simple and portable form of object-based so that it can be easily understand, modify and debug than its sequential counterpart. These PPL should have constructs which must be capable of exploit the level of parallelism inherently present in the application matching the underlying architecture (ISA of the node architecture).

2.2 Towards PPL catering heterogeneous paradigm

Thus the thesis proposes a significant breakthrough in programming model by proposing WPPL: Warft Parallel Programming Language, a Object-oriented higher level assembly constructs, by defining a level of abstraction at a point where both architecture and programmer which helps in harnessing the highest degree of parallelism exhibited by the application and reduces the burden on compiler, thus significantly increasing the instruction generation rate by many folds to cater to the thousands of CPU/GPU cores at the cluster level. Our results show that, by adopting to the proposed programming model achieves a linear reduction in CPU cycles and memory access operations for the same problem size, thus directly contributing to the reduction in energy consumption compared to the conventional higher level programming models. We also demonstrate that our parallel programming framework is able to significantly adapt to the changes in input problem size and system architecture.

Recent trends clearly points out that the future of computing is towards heterogeneous compute resources in order to achieve optimal resource utilization [?] [14]. There are many tools available for the automatic mapping of sequential programs on homogeneous multi-core architectures. But most of the tools or languages are inefficient in catering heterogeneous architectures. Hence a level of abstraction must be
introduced with in the comfort levels of the programmer, thus the programmer keeps the underlying architecture in mind and program accordingly. This knowledge helps the programmer to actually know the exact potential what his architecture is capable of and hence program them accordingly to make use of them its fullest.

2.3 Bridging the gap between the application and architecture

A C++ or Java programmer would probably take the raw arrays and loop patterns and wrap them in a class in order to abstract the underlying data structures away from their uses, providing a vivid interface for accessing and iterating. With good naming choices, this OOP-based approach can go a long way toward making the code more programmable and comprehensible to a human reader. Yet for the compiler, it does little to help, and often hurts, by adding more code framework to sort through in analyzing the computation (including the potential for dynamic dispatch issues if the class is part of a larger matrix class hierarchy). Such examples of higher-level programming are arguably a large part of why the HPC community tends to conflate productivity with poor performance.

Considering scalable parallel programming, perhaps the most obvious languages to extend are C/C++. C and C++ emerged from more of a systems programming setting and betray this through concepts like pointer-array equivalence and lack of rich support for multidimensional arrays. Arguably these choices have prevented C/C++ from completely replacing FORTRAN in the scientific community. They also make C/C++ a difficult starting point for languages designed for large-scale scientific computation.

Thus the proposed PPL paradigm language is very close to the architecture i.e
higher level assembler and hence it becomes more easier for the compiler to generate efficient code to program the hardware. The model proposed should also be capable of maintaining concurrency and aids in efficient communication across cores/nodes. WPPL constructs are also intended to facilitate compilation more effectively, as the onus laid on the compiler is greatly reduced.

2.4 Algorithm based parallel execution model

Every other model has its own way of extracting parallelism Viz. thread based, process based, task based, etc., WPPL extracts algorithm level parallelism, because of the inherent design of the ALFU (Algorithmic Level Functional Units) [3]. Since the CUBEMACH [1] design paradigm uses ALFU to accelerate the scientific libraries, WPPL extracts the parallelism that is dormant in an algorithm. Hence WPPL could actually cater to a wide range of scientific libraries and simulations as the major portion of the applications are mutable in the form of algorithm. This is an important advantage of the proposed WPPL paradigm, which supports different classes of applications.

If a language could support sparse matrices or arrays directly, and are also architecture aware, such a language could provide similar productivity benefits to the user as the OOP approach, and often improve upon it, due to the opportunity to support a specialized syntax. As a result, the compiler can shift its focus from heroically wrestling with optimizing indirect indices and unraveling method calls toward to issues that are more closely related to the semantics that the programmer wanted to express anyway. Thus, the end-user gets improved programmability while the compiler gets more semantic information to use in performance optimizations a win-win situation.
Conventional HLL parallel constructs need to be vastly re-framed to bring in power and performance efficiency and to reflect the architecture hardware for better resource utilization and to ensure much reduced execution complexity. While trying to develop parallel constructs, unlike the existing, they should be user friendly to a large extent although these constructs are built to reflect the underlying architecture hardware. When these constructs directly reflect the underlying hardware architecture power and performance efficiency is better achieved, and more so if the underlying architecture is composed of higher level units (ALFUs). In such case, the parallel constructs cannot be at higher level of abstraction.

2.5 Integrated Constructs for Computation, Communication & Parallelism

The proposed WPPL has integrated constructs for computation to execute applications on the parallel architectures. The parallel looping constructs that operates on a set of data parallely ensures high degree of parallelisation of applications across cores. WPPL has dedicated constructs for communication, which act upon the network traffic model, which also ensure efficient data placement. Specialized instructions for control and exceptions make sure that the errors are handled by the program by itself. Thus WPPL overall encompasses the entire constructs that addresses the performance and power issues of parallel programming paradigms.

WPPL is basically composed of lower level constructs. For such low level constructs, unlike HLL compilers, for example the current day assemblers do not use any code optimization process. People have not researched in to these aspects of assembly compiler code optimization and evolving parallel user-friendly assembly like constructs and if successful it could be far more efficient than conventional HLL
parallel constructs and the associated compilers.

**Figure 2.2:** Warft Parallel Programming Language Execution Flowchart

This thesis tries to investigate in to the above aspects of developing user friendly low level assembly like parallel constructs and code optimizing assembler (assembler can be called assembly compiler in the sense of HLL) reflecting the underlying architecture. Hence application developed using such low level parallel constructs is bound to reduce the power drastically and also increase the performance levels in supercomputing clusters. Thus the assembly like parallel programming constructs which is well-established language with rich support for arrays, object-oriented programming, generic programming, iterator functions, and type inference and also evolving data structures that suite high performance applications with an open-source optimized assembler support is the ultimate goal of the thesis.

### 2.6 WPPL Execution Flow

The Figure[2.2] captures the entire flow of the WPPL: Warft Parallel Programming Language where in, the application is programmed using the proposed WPPL constructs with the knowledge of the underlying architecture. Thus the WPPL
Table 2.1: Looping and Serialization Constructs

takes advantage over the conventional HLLs especially because of its attributes likes looping constructs, communication constructs, support for multi-dimensional data structures for complex scientific libraries along with control and exception constructs. Thus the WPPL is compiled in the WPPL compiler which generates the ALISA (Algorithm Level Instruction Set Architecture) libraries for CUBEMACH paradigm [1].

Thus the following parallel constructs have been evolved keeping the CUBEMACH design paradigm in mind. Thus the algorithm constructs are programmed in a way such that it minimizes the communication across nodes and also taking advantage of the CUBEMACH’s design paradigm. Generated Assembly Level Logic Instruction Sample:

#APPID #NODEID #INSTID #UNIT-TYPE #DATAMEM1 #DATAMEM2 #DESTINATION_ADD
<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>The memory partitioning schema of the application</td>
</tr>
<tr>
<td>Hybrid</td>
<td>Combination of shared and distributed memory</td>
</tr>
<tr>
<td>Fully_Distributed</td>
<td>Fully distributed memory schema</td>
</tr>
<tr>
<td>Fully_Shared</td>
<td>Fully shared memory schema</td>
</tr>
<tr>
<td>Mode</td>
<td>Mode based code generation</td>
</tr>
<tr>
<td>Performance</td>
<td>Maximum parallelization mode</td>
</tr>
<tr>
<td>Power</td>
<td>Minimum communication mode</td>
</tr>
<tr>
<td>Atomic Instructions</td>
<td>Supports atomic operations</td>
</tr>
<tr>
<td>Read(i)</td>
<td>Read the current value</td>
</tr>
<tr>
<td>Write(i)</td>
<td>Write the current value</td>
</tr>
<tr>
<td>Logic(Add[i &amp; j])</td>
<td>Perform atomic logical operations</td>
</tr>
<tr>
<td>Clear(i)</td>
<td>Clear the current value</td>
</tr>
</tbody>
</table>

Table 2.2: Model, Mode, Atomic Constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(Data)</td>
<td>Sending the data packet from one node to another</td>
</tr>
<tr>
<td>Recv(Data)</td>
<td>Receiving the data packet from one node to another</td>
</tr>
<tr>
<td>Asend(Data)</td>
<td>Asynchronous data send</td>
</tr>
<tr>
<td>Broadcast(Data)</td>
<td>Sending data to all the connected nodes</td>
</tr>
<tr>
<td>Wait &amp; send</td>
<td>Wait for acknowledgment and send</td>
</tr>
<tr>
<td>Buffer &amp; routing controls</td>
<td>Intermediate buffer and routing protocols</td>
</tr>
<tr>
<td>Communication aware construct</td>
<td>Facilitates optimal data movement and placement</td>
</tr>
<tr>
<td>Glue construct</td>
<td>Resolve the data incompatibility issues in heterogeneous</td>
</tr>
<tr>
<td>Synchronization constructs</td>
<td>Maintains the Sync across cores</td>
</tr>
<tr>
<td>Auto_Sync(PiD)</td>
<td>Automatically sync instructions</td>
</tr>
<tr>
<td>Manual_Sync(Var)(PiD)</td>
<td>Manually sync instructions</td>
</tr>
</tbody>
</table>

Table 2.3: Communication and Synchronization Constructs
<table>
<thead>
<tr>
<th>Constructs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD(LiB)</td>
<td>Single Program Multiple Data mode suppport</td>
</tr>
<tr>
<td>Halt(PiD)</td>
<td>Halt till the flag is reset</td>
</tr>
<tr>
<td>Nop(PiD)</td>
<td>No Operation</td>
</tr>
<tr>
<td>Lock(PiD)</td>
<td>Lock till the flag is reset</td>
</tr>
<tr>
<td>Wait(PiD)</td>
<td>Wait till the flag is reset</td>
</tr>
<tr>
<td>Exception constructs</td>
<td>Resolving Exceptions of all kinds</td>
</tr>
<tr>
<td>Buffer_Overflow</td>
<td>Warns the scheduler in case of buffer overflow</td>
</tr>
<tr>
<td>Deadlock_On_Resource</td>
<td>Deadlock detector and resolver</td>
</tr>
<tr>
<td>Logic_Error_Exception</td>
<td>Divide by zero errors</td>
</tr>
<tr>
<td>GLUE_Computer_Exception</td>
<td>Data incompatibility resolving error</td>
</tr>
<tr>
<td>Unresolved_Network_Traffic</td>
<td>Unresolved network blocking</td>
</tr>
<tr>
<td>DNF_Error</td>
<td>Data not found error</td>
</tr>
<tr>
<td>Synchronization_Errors</td>
<td>Unresolved sync issues</td>
</tr>
<tr>
<td>Coherence_Error</td>
<td>Data coherence errors</td>
</tr>
<tr>
<td>TSP_Exception</td>
<td>Time stamp exception</td>
</tr>
<tr>
<td>Ctrl_exception</td>
<td>Control exception</td>
</tr>
<tr>
<td>Data_Decode_Conflict</td>
<td>Exception occurred during data decode</td>
</tr>
</tbody>
</table>

**Table 2.4: Control and Exception Constructs**
3.1 **PPL Model Evaluation**

Three different classes of algorithms (Viz., numeric, semi-numeric, non-numeric) were taken up for the initial study to validate the investigation. On comparing several higher level programs with our programming paradigm, we could clearly observe a performance difference in the execution times and memory accesses. The Initial pilot simulation and study was carried out on i3 dual core x86 Nehalem machine. Further investigation is intended to be carried out in a cluster environment to tap the inherent parallelism from the cluster.

3.2 **Execution Time**

It could be clearly observed in the Fig [3.1] that the execution times in terms of clock cycles for the algorithms coded in C++ is two to eight times higher than that of the programs coded in WPPL. The trend continue to scale up as the problem size increases.

3.3 **Cache access**

From the Fig [3.2] for varying problem sizes the L1 and L2 cache accesses are plotted. It is clear that the WPPL generated code restricts the data movement and access by generating efficient machine code. Thus this directly contributes to the energy efficiency due to restricted data movement.
Figure 3.1: Execution time comparison

Figure 3.2: Cache Access Comparison
3.4 WPPL Code Snippets

Three different classes of algorithm study is been undertaken and the snippets of the algorithm coded in WPPL are given in the Fig[3.3]. It could be noted that coding in WPPL improves the programmer productivity and reduces the burden laid on the compiler or code generator.

```wppl
//WPPL CODE
//LU-DECOMPOSITION
MODEL: Hybrid
MODE : Power
INT MATRIX[][];

INIT<ARCHITECTURE.XML>

Logic
{
PRINT<Enter Number of Vertices>
GET>>X;
FOR_WINDOW_SLICING_2D [i..n || j..k]
GET<MATRIX [i][j]>
LUD_UNIT<MATRIX>
}

Communication
{
X=LUD_UNIT<Query>
DATA_PARTITION<X>
SEND<MAP-DATA-NODES>
CALL {LOGIC} \ Program logic
RECV<COLLECT-REDUCE-DATA>
PRINT<RESULTS>
}

//WPPL CODE
//BREADTH FIRST SEARCH
MODEL: Fully_Shared
MODE : Performance
INT VERTICES[];

INIT<ARCHITECTURE.XML>

Logic
{
PRINT<Enter Number of Vertices>
GET>>X;
FOR_UNROLL[1..n]
GET<VERTICES>
BFS_UNIT<VERTICES>
}

Communication
{
X=BFS_UNIT<Query>
DATA_PARTITION<X>
SEND<MAP-DATA-NODES>
CALL {LOGIC} \ Program logic
RECV<COLLECT-REDUCE-DATA>
PRINT<RESULTS>
}

//WPPL CODE
//SELECTION SORT
MODEL: Fully_Distributed
MODE : Performance
INT ELEMENTS[];

INIT<ARCHITECTURE.XML>

Logic
{
PRINT<Enter Number of Elements>
GET>>X;
FOR_UNROLL[1..n]
GET<VERTICES>
BFS_UNIT<VERTICES>
}

Communication
{
X=SORT_UNIT<Query>
DATA_PARTITION<X>
SEND<MAP-DATA-NODES>
CALL {LOGIC} \ Program logic
RECV<COLLECT-REDUCE-DATA>
PRINT<RESULTS>
}
```

Figure 3.3: WPPL Snippet Codes for Different Classes of Algorithms
CHAPTER IV

WORK TO BE COMPLETED

4.1 WPPL Compiler

The Host Compiler for generating the machine code is the work to be taken, followed by incorporating various code generation and optimization techniques, which are pruned for heterogeneous architecture units.

4.2 Network Processor constructs

Further the assembly constructs for the network processor is yet to be conceived. Once the constructs for network processor is defined, then the data movement and placement constructs become much more effective. Thus the constructs would be traffic-aware and the data placement would be based on analytical traffic model equations.

4.3 WPPL based Application Execution in WIMACS

Execution of WPPL on WIMACS: Warft India Many Core Simulator and analyzing the influence of the model and its constructs in the cluster environment. Detailed study of the generated code and compiler efficiency, thus bringing out its advantages over other Higher Level Language’s and its behaviour in cluster. [18]
4.4 Porting Scientific Libraries

Several scientific dense libraries like LINPACK, LAPACK, BLAS, QRD, Cholesky and other signal processing libraries which includes DFT, FFT needs to be ported to the proposed programming paradigm and a thorough investigation must be conducted in the aspects of performance and power efficiency including memory accesses.
REFERENCES


[6] Partitioned Global Address Space Programming Model
https://sites.google.com/a/lbl.gov/pgas12/


[16] Jithin Jose, Miao Luo, Sayantan Sur, Dhabaleswar K. Panda, 'Unifying UPC and MPI Runtimes: Experience with MVAPICH'

[17] Ahmed A.Jerraya, et.al., 'Programming models and HW-SW Interfaces Abstraction for Multi-Processor SoC'

