ABSTRACT
Driven by the ever-growing demand for computing power, computers are becoming more and more powerful. However, in recent years, due to the physical limitations, this increased computing power does not come in the form of increased CPU clock speed, but in the form of more cores (processors) in a single chip die. Computer industry has started to use this new multi-core technology to massively produce systems for both stand-alone desktop PCs and high-end servers. In the near future, multi-core cluster will become one of the most economic supercomputer architectures. In order to utilize the full power of multi-core systems, some kind of parallel computing is necessary. However, parallel programming is notoriously known as a challenge job. This paper analyzes different parallel programming models, compares their strengths and weaknesses on multi-core based systems, and introduces an on-going project on providing a better parallel programming environment based on a novel View-Oriented Parallel Programming (VOPP) model.

Categories and Subject Descriptors
D.3.2 [Software]: Programming Languages—Concurrent, distributed, and parallel languages

Keywords
View-Oriented Parallel Programming, Multi-core system, Parallel Programming, MPI, OpenMP, Pthreads

1. INTRODUCTION
Dating back to 1958, parallel computing is not a new idea [15]. However, up until now, it mostly concerns only people from high performance computing (HPC) sector. This situation is now changing. In recent years, since there are a number of physical limitations, like heat generation, CPU clock speed stops to grow. But the Moore’s Law still holds, which means there still exists the ability to pack more transistors on a single chip. Therefore, the recent trend in microprocessor industry is to put more cores (processors) into a single chip die, which is also known as the multi-core technology. Unlike in the old architecture (von Neumann architecture), where applications enjoy speedup by simply running on a newer generation CPU, in systems based on multi-cores, virtually all existing software will not take advantage of the extra cores available if not modified. This is where parallel computing plays an important role.

Yet parallel programming is commonly known as difficult for at least the following reasons:

- It adds complexity to the problem at hand to spot and decompose the parallelism.
- It is error-prone and harder to debug, because programs are executed concurrently on many computing units and thus it is not easy to reproduce the situation where the error occurs.
- It is difficult to tune the performance, because high performance depends on a wide range of factors, e.g. characteristics of the application, granularity of parallelism.
- It lacks a higher level abstraction in the parallel programming languages. Most of the existing programming models are somewhat low level, which takes the programmer closer to the hardware and farther from the application domain.

Most of above problems are very difficult to solve, if at all possible. Yet, efforts are constantly being made to relieve the problems.

This project aims to design and implement a parallel programming system on a multi-core cluster based on View-Oriented Parallel Programming (VOPP) [11], as well as to extend the API of VOPP in order to facilitate VOPP programmers.

The rest of the paper is organized as follows: Section 2 presents the hardware of different parallel computing platforms in the past and at the moment, and compares the major parallel programming models on these platforms; Section 3 introduces the View-Oriented Parallel Programming model, then it analyzes its advantages over other models; Section 4 describes the project; testing and evaluation are discussed in Section 5; and finally project status and future work are given in Section 6.
2. BACKGROUND
There is a wide range of parallel computing platforms and parallel programming models. Some models work better on some platforms. The following sections will first discuss the different hardware, and then the different programming models and their respective strengths and weaknesses.

2.1 Hardware
A parallel computing platform is one that contains multiple processing units (sometimes called execution units). There are two main classes: tightly-coupled multicomputers and loosely-coupled multicomputers. In the early days of high performance computing, special mainframe supercomputer were built for specific applications. They have multiple identical processors and share the same memory address space. It is also termed as a symmetric multiprocessor (SMP). The SMP is one typical kind of tightly-coupled multicomputer. Later, when Personal Computers (PC) became powerful and cheap, a number of PC is grouped together through a private network to form a cluster, which is a typical kind of loosely-coupled multicomputer. CPU in each node has its own local memory. Now, there is multi-core system, which essentially is a number of CPUs packed into a chip and falls to the tightly-coupled multicomputer category. It can be viewed as a small cluster, except that all cores have access to a shared memory address space. Figure 1 and Figure 2 show the differences between these two classes. There are also other newly released hardware like GPGPU [1].

2.2 Explicit Threads
POSIX Threads (Pthreads) is one of the most used thread models for explicit threads. It is used in Linux and some other Unix systems. Other models include .NET threading API for Windows and Java multithreading for JVM. Pthreads provide API for programmers to manage thread creation, destruction, and management (including synchronization). Parallelism is usually expressed by creating multiple threads, with each thread processing part of the problem, which could be in the form of data decomposition or task decomposition.

A sample code is as follows:

```c
void *thread_func1(void *args1); void *thread_func2(void *args2);
int shared;
pthread_mutex_t sharedMutex;
sharedMutex=PTHREAD_MUTEX_INITIALIZER;

int main(void){
    int i , j;
    pthread_t thread1 , thread2;
    pthread_create(&thread1 , NULL,
                   thread_func1 , (void*)&i);
    pthread_create(&thread2 , NULL,
                   thread_func2 , (void*)&j);
    pthread_join(thread1 ,NULL);
    pthread_join(thread2 ,NULL);
    exit(EXIT_SUCCESS);
}

void *thread_func1(void *args1){
    pthread_mutex_lock(&sharedMutex);
    // update shared
    pthread_mutex_unlock(&sharedMutex);
    // do some computation
```

Figure 1: Tightly-coupled Multicomputers

Figure 2: Loosely-coupled Multicomputers
In the above code, two threads are created, with each of them doing part of the computation of the problem at hand, and later join together. There is also a shared data between the two threads, and a mutex is used to protect access to the shared data.

Explicit threads like Pthreads provide direct control over thread management. Therefore programmers have significant flexibility. However, the advantage of it is also its disadvantage. As shown in the sample code, programmers are burdened with all the details, like thread creation, locking, etc. It is easy to get data race condition (i.e. when two or more threads are accessing the same data, with at least one is a write access), or deadlock errors. A handful of threads could be easily manually managed by programmers, however, when the number increases to, say, 50 threads, it is no longer a trivial job to coordinate different threads. A better way is needed so that thread management is transparent to programmers.

2.3 OpenMP

OpenMP [2] is developed as a higher level abstraction of thread model, which means at its core, OpenMP uses threads, but the details are hidden from programmers. As with all higher level approaches, OpenMP sacrifices flexibility for the ease of writing code.

OpenMP is implemented as a collection of compiler directives and library routines. It can be used with base language C/C++ or Fortran. A typical use of OpenMP is to parallelize computationally heavy loops by augmenting the loop with a simple OpenMP directive. Compiler will use this directive to automatically create, synchronize, and destroy threads.

A sample code is shown as follows:

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
    int N=20000;
    int i, a[N];
    // point A
    #pragma omp parallel for
    for (i=0;i<N;i++)
        a[i]= i*3;
    // point B
    return 0;
}
```

The above code tells the compiler to execute the for loop in parallel, in which case the program will at point A fork a number of threads (specified by environment variable or by OpenMP library routine) and after the execution, all threads are join together at point B, and the program will continue to execute the following statements, as shown in Figure 4.

A serial version of a program for a problem is usually easier to write than a parallel one. So an advantage of this type of approach is that it may be possible to get a parallel program by simply adding some OpenMP directives to a serial program. However, we must also notice that for loops containing data dependency, special care is needed to remove data race, e.g. having a local copy of those data, using locks, or critical regions.

With the advent of multi-core systems and the progress made in standardization, OpenMP becomes very popular recently. However, one major problem of OpenMP is scalability. Since it is based on a shared memory machine, it is confined by the available computing units in the system. So when the problem size increases, it is hard to add more resource to speed it up. Although research is going on to apply OpenMP on cluster computers, like Cluster OpenMP by Intel [8], the result is not satisfactory.

2.4 Message Passing Interface (MPI)

In the High Performance Computing (HPC) sector, MPI is often a de facto standard for loosely coupled multicomputers. In contrast to threaded approaches, MPI uses messages to copy memory from one process space (program) to another.

MPI is a set of library functions in conjunction with base language of C or Fortran. A sample code is shown as follows:

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
    int my_rank; // process id
    int p_num; // num of processes
    #include <stdio.h>
    #include <string.h>
    #include "mpi.h"

    main(int argc, char* argv[]) {
        int my_rank; // process id
        int p_num; // num of processes
    }
```
3. VIEW-ORIENTED PARALLEL PROGRAMMING (VOPP)

VOPP is another parallel programming model, which is based on shared memory or distributed shared memory. It contains a user-level library (used for view management) and is used in conjunction with the C programming language.

In VOPP, shared data is partitioned into views. A view is a set of memory units (bytes or pages) in (distributed) shared memory. Each view, identified by a unique view_id, can be created, merged, and destroyed at any time in a program. Before a view is accessed (read or written), it must be acquired (e.g., with view primitive acquire_view); after the access of a view, it must be released (e.g. with view primitive release_view). The most significant property of views is that they do not overlap with each other.

The following classes of views are identified in [9] for parallel programming: Single-Writer View (which includes Consumable View and Atomic View), Multiple-Writer View, and Automatically Detected View.

Unlike OpenMP or Pthreads, VOPP uses processes instead of threads. A sample code is shown as follows:

```c
int * sharedArray;
int * localArray;
int arraySize;

main(int argc, char **argv){
    int i, j, start, end;
    long sum;
    arraySize=10000;
    // VOPP startup
    Vdc_startup(argc, argv);
    if (Vdc_proc_id==0) {
        shared_array = 
        Vdc_malloc(size*sizeof(int));
        // init sharedArray
    }
    localArray=malloc(sizeof(int));
    // init local array
    for (i=0;i<Vdc_nprocs;i++) {
        start=(i+Vdc_proc_id)%Vdc_nprocs /
        *size/Vdc_nprocs;
        end=((i+Vdc_proc_id)%Vdc_nprocs+1)
        *size/Vdc_nprocs;
        Vdc_acquire_view((i+Vdc_proc_id%
        Vdc_nprocs);
        for (j=start;j<end;j++)
            sharedArray[j]=localArray[j];
        Vdc_release_view((i+Vdc_proc_id%)\n        Vdc_nprocs);
    }
    Vdc_barrier(0);
    if (Vdc_proc_id==0) {
        for (j=0;j<Vdc_nprocs;j++)
            Vdc_acquire_view(j);
        for (i=size-1;i>=0;--)
            sum += sharedArray[i];
        for (j=0;j<Vdc_nprocs;j++)
            Vdc_release_view(j);
    }
}
```

In the code, every process has its local array and needs to add it to a shared array. The shared array is divided into Vdc_nprocs equally-sized views. As shown in the example, all view accesses are bundled with view acquire and release primitives. With a novel View-based update protocols [12], the bundling of mutual exclusion and data access does not degrade its performance, however, mutual exclusion is automatically achieved and data race condition is then relieved from programmers.

There are also other programming interfaces that bundle mutual exclusion and data access [7, 13]. For example, CRL (C Region Library) [14] focuses on low-level memory map-
ping, and limits a region to contiguous memory space. In contrast, a view in VOPP is a higher level shared object whose memory space may be non-contiguous, e.g., Automatically Detected Views. Entry Consistency (EC) [7] and Scope Consistency (ScC) [13] also bundle mutual exclusion and data access like in VOPP. However, their programming interfaces are very different from VOPP [10].

4. MY PROJECT
As the multi-core technology advances, multi-core/many-core systems will become cheaper and more powerful. So multi-core/many-core clusters will become one of the most economic parallel computing platform. However, this platform is a combination of shared memory and distributed memory. To make an efficient and easy to use parallel programming environment on such platform, the underlying system must differentiate memory hierarchy yet providing an interface that hides the details from programmers.

To use MPI on a multi-core cluster has two disadvantages. First, although it is efficient to use MPI on distributed memory systems, it is less efficient to use it on shared memory systems (i.e. within a node of a multi-core cluster). Second, MPI is harder to program and less straightforward than shared memory based programming model.

Some proposals to mix MPI and OpenMP for faster execution on a multi-core cluster. This kind of hybrid programs can in many situations improve performance, however mixing two different programming model obviously makes parallel programming even harder.

Therefore a new programming environment is needed for multi-core clusters. VOPP has a number of advantages over both MPI and OpenMP, since the view concept from VOPP is a general idea. At this moment, VOPP has been implemented on multi-cluster systems, called the VODCA implementation. Its performance is comparable to MPI [10]. VOPP has also been implemented on multi-core computers and is called Maotai. Since VOPP has demonstrated its promising performance on both cluster computers and multi-core computers, it is reasonable to make the hypothesis that VOPP can deliver good performance on a multi-core cluster.

However VOPP is not mature yet. Only very basic and low level API has been proposed and implemented. The difficulty in parallel programming is largely due to the lack of a high level abstraction in parallel programming languages. This resembles the situation when assembly language was used to program, and that led to a low programming productivity. Later, higher level programming languages, like structure languages and object-oriented languages, were introduced, which greatly improved programming productivity. Therefore, in order to address the software development crises on parallel platform, we need to design a parallel programming language with high level abstraction, which makes it easier to model the problem at hand.

4.1 The problem statement
This project is to design and implement a parallel programming environment on multi-core clusters based on View-Oriented Parallel Programming, as well as to design and implement a set of VOPP API to facilitate programmers writing parallel applications.

4.2 Difficulties
To design a set of high level APIs, one must have a sound understanding of the characteristics of different parallel applications, such as the seven applications shown in [6]. It is hard to justify the design by just thinking. One must do so by using it. When implementing those API, an efficient way of allocating computation to different computing units according to its location is needed.

5. TESTING AND EVALUATION
When the new API is designed, to demonstrate whether it is general, expressive, and efficient, we need to make thorough testing using standard parallel computing benchmarks, namely the High-Performance Computing Challenge (HPCC) benchmarks suite [3] and the NAS Parallel Benchmarks(NPB) suite [4]. For programmability comparison, one objective criterion is to compare the number of lines of the applications written in VOPP and MPI/OpenMP. Subjectively, we can also compare programming styles and its suitability to different parallel programming patterns.

When the new API is implemented, its performance needs to be evaluated. We will compare the running time of the benchmark applications written in VOPP and MPI/OpenMP. Speedup and scalability are two most important factors for parallel computing platform.

The design and implementation phases need to go concurrently, as performance tests can give feedback to the design. The performance tests need to run on different platforms. At this moment, we have access to the following platforms: A cluster with 32 PC nodes; A Sun T1000 server with UltraSPARC T1 [5] processor, which contains 8 cores; A multi-core cluster with four Sun T2000 server, each of which contains a UltraSPARC T1 processor.

6. STATUS AND FUTURE WORK
The project started in the second semester 2007. Literature review is still in progress. I have also began reading source code of initial VOPP implementation on clusters and started to program in VOPP for small problems to get familiar with the environment. Future work includes more source code reading of the current implementation of VOPP API, comparing different programming styles, design new API, implement them, and test their performance.

7. ACKNOWLEDGMENT
Thanks to Syed Faisal Hasan, a senior PhD student in our department, for encouraging me to participate in this student conference.

8. REFERENCES
62


