ORACLE
SOLARIS

Developing Parallel Programs - A Discussion of Popular Models

ORACLE WHITE PAPER | MAY 2016

ORACLE

Table of Contents

Introduction

This white paper discusses emerging issues and key trends in the area of parallel software development, and examines several common approaches to the process of parallel programming. Common industry frameworks, APIs, and standards, such as Open MultiProcessing (OpenMP), Intel® Threading Building Blocks, Apache Hadoop, Message Passing Interface (MPI), and Apple's Grand Central Dispatch, are described and illustrated using a typical programming example. This white paper explores the outlook for parallel software development and focuses in particular on application development for shared memory multicore systems. Two real-world application case studies are introduced to highlight some challenging parallel program design issues. Important tools critical to parallel software design phases also are discussed along with the pros and cons of each tool.

Overview

With the ubiquity of multicore processors and other recent advances in computer hardware, parallel computing is emerging as a vital trend in mainstream computing. Two other significant computing developments — cloud computing and service-oriented architecture (SOA) — combined with parallel computing comprise three critical areas that are heavily influencing application software development now and for the foreseeable future. While seemingly unrelated, these three areas follow an evolutionary path from earlier technologies and are tightly intertwined. As a result, it is intriguing to observe and predict how the course of application software development progresses in conjunction with these crucial technological trends.

Of the three trends, the most widely adopted is SOA, coupled with the popularity of dynamic languages. An evolution of the client-server model that has been around for well over a decade, SOA [1,2] imposes a relatively clean and structural way to develop and deploy enormously complex business software. Derived from powerful modern platforms, the concept of SOA assumes scalability when the demand for services increases. However, that scalability often translates into expensive and complex software and hardware systems.

Today's new and faster multicore and chip multithreading processor designs are making scalability more affordable. Processor technologies with near zero latency interconnects, coupled with new Web service standards, are spurring the adoption of SOA application software. As demand for scalability becomes more pervasive, it is critical for developers to design scalable parallel applications to take advantage of the latest technology.

Cloud computing [3] is the newest of the three important technology trends and offers parallel programming possibilities to the application software industry. One of the hottest buzzwords in the IT industry today, cloud computing has evolved from client-server and SOA models due to major breakthroughs in internet bandwidth that make it possible to take a giant leap in providing computing power, platforms, infrastructure, or software as a service. With cloud computing, programmers can easily access the powerful compute capabilities previously available only to national laboratories and technology centers.

Another important technology trend that is closely related to parallel and cloud computing is grid computing [18, 19, 20]. As technology improves, users are transforming the internet from a means of passive information sharing to active service and resource sharing. This technology makes the computing power circulate like currency to be utilized in the optimal way.

Advanced scientific and engineering communities have long used parallel computing to solve large- scale complex problems, but even these developers find it hard to implement parallel applications effectively. To aid the development of parallel applications, several programming models have been created. The most frequently used ones are OpenMP [4, 5] for shared memory programming, and MPI [6, 7] for distributed memory programming. Because of the growing number of multicore machines being brought to market, this paper focuses more on the shared memory model than the distributed computing model.

A recent innovation is the use of run-time libraries for shared memory architectures. Intel offers the Threading Building Block (TBB) [8], while Microsoft offers the Task Parallel Library (TPL) [9] and Parallel Patterns Library (PPL) [10]. In June 2009 [11], the Oracle Solaris Studio product team presented a new parallel C++ run-time library called MCFX Framework (later renamed to Parallel Framework Project) at the International SuperComputing Conference in Hamburg, Germany.

Generally, parallel programming models or frameworks are based on either language enhancement or run-time libraries. This technical white paper discusses OpenMP (language enhancement), Intel TBB (run-time libraries), Parallel Framework Project, Map-Reduce [12] and Hadoop [13], MPI, and Grand Central Dispatch [14] with C/C++ block extension. Although there are other important programming models such as Unified Parallel C (UPC) [15],

Open Computing Language (OpenCL) [16], and Compute Unified Device Architecture (CUDA) [17], only the selected models above are used to explore the current state of parallel application development.

The case studies in this paper examine two challenging parallel design issues found in real-world applications — searches and simulations. These vital programming features are widely used in various academic and industrial applications. Both case studies exemplify the difficulty of developing parallel applications even given the current advanced state of parallel programming technology.

There is a well-known ancient Chinese saying: “A craftsman first sharpens his tools before laboring on his work.” This is very true of software development, particularly in the area of complex parallel applications. Parallel programming presents many pitfalls — race conditions are the most common and difficult multithreaded programming problem. Overhead due to thread synchronization and load balancing can severely impact run-time performance and can be very hard to fix, humbling even the most competent programmer. In addition, many experienced software developers are more afraid of implementation faults such as memory errors than bad programming logic. These pitfalls of parallel application development exemplify the difficulty of implementing parallel programs.

The last section of this white paper discusses several important tools that can be used in various stages of the software development cycle to aid programmers in parallel application development. Rather than offering a judgment on which parallel programming model is better or worse, it is hoped that this overview and comparison of the various frameworks can help developers to select the appropriate tools. With the information presented here, developers can decide which model is better suited to a particular project and evaluate the tools according to their own personal tastes and preferences.

An Example of Parallel Programming

```
typedef std::map<const char*, int, StrLess> word_map_t;

class parser_obj { // parse input text file for "line_num" lines.
public:
    static int line_num;
    int line_cnt;
    word_map_t wmap;
    char** lines;
    parser_obj* prev;

    parser_obj()  {
      line_cnt = 0; prev = NULL; lines = new char*[line_num];
    }

    void process() {
       ParseInputTextIntoWords();

       For each  cur_word in ListOfParsedWords {
         word_map_t::iterator hit = wmap.find(cur_word); // is it already in wmap?
         if (hit != wmap.end()) {
             (*hit).second = (*hit).second + 1;    // yes, increment count
         } else {
             word_map_t::value_type new(cur_word, 1);
             wmap.insert(new); // no, enter new
         }
       }
    };
};
```

Figure 1. A program example of the parser_obj class

A variant of the canonical wordcount program is used as an example of how to write a parallel program. This program is used in many documents that describe Google's MapReduce [12] design pattern (discussed in the MapReduce section), and the usage of other parallel programming models such as OpenMP, TBB, and Parallel Framework Project (PFP) are shown in subsequent figures by implementing the same wordcount program with the particular features of each framework.

The sample program reads a large text file line by line and counts the number of occurrences for each unique word in the file. The program counts the total number of unique words and indicates which word in the file occurs the most frequently. The lines of the input file can be divided among many threads or processes, providing ample opportunity for parallelism to speed counting. The trick is being able to combine the results of the multiple threads efficiently in order to arrive at the final result. While the original code is written in C++, the example programs use C++ mixed with pseudo-code for the sake of brevity.

Some common data structures are used in implementing the wordcount example in all of the aforementioned frameworks. Good programming practices express the inherent parallelism of the program by using data structures that are portable across multiple frameworks with little modification. Separate chunks of source data that can be worked on concurrently by multiple threads or tasks are encapsulated within an object-oriented design representation. A good practice in single-threaded programming, this approach easily lends itself to parallel execution.

Since the input file can be fairly large, the program should parse it in parallel. The example creates several objects of the class `parser_obj`, which is responsible for parsing multiple lines of the input files into strings (Figure 1). The parsed strings are stored in a *word map* (using the standard template library map construct) that also keeps track of the number of occurrences of each word.

Another key data structure is the `collect_obj`, which combines the intermediate results generated by the parser objects (Figure 2). The example creates an object of class `collect_obj` for each letter of the alphabet and let these 26 collecting objects scan through every parser object and its corresponding `wmap` data. It skips letters not starting with each collect object's specified letter and combines the cumulative word count for each object in the `cmap` data. Depending on the framework, one can create a collect object for each of the available threads and allow each collect object to gather data for a range of letters of the alphabet. For example, the first collect object may look at words starting with the letters 'a' through 'e', the second from 'f' through 'j', and so on.

```
class collect_obj {
public:
     int diff_word_count; total_word_count; max_word_count;
     const char* max_word_str;
     word_map_t cmap; // Store Combined result of all words starting with
             // particular letter from all parser_object wmaps;
     int alpha_index;
     parser_obj* start_pobj;
     collect_obj(int idx, parser_obj* pobj) {initialize};
     void process();
};

void collect_obj::process() {
  char lead_char = 'a' + alpha_index;
  parser_obj* pobj = start_pobj;
  while (pobj != NULL) {
   word_map_t::iterator wt;
   for (wt = pobj->wmap.begin(); wt != pobj->wmap.end(); wt++) {
      char *cur_word = (*wt).first;
      if (cur_word[0] != lead_char) continue;
      if ((cur_word already in cmap) update word_count in cmap;
      else enter cur_word into cmap;
      Update total_word_count, diff_word_count and max_wordcount;
   }
   pobj = pobj->prev;
 }
}
```

Figure 2. A program example of the collect_obj class

OpenMP

OpenMP [4, 5] is an industry standard API design specification developed through the joint efforts of top computer manufacturers and software developers such as Oracle, Hewlett-Packard, IBM, and Intel. It is supported on the most widely used native programming languages such as Fortran, C and C++. OpenMP offers a common specification that lets software programmers easily design new parallel applications or parallelize existing sequential applications to take advantage of multicore systems configured with shared memory. For more detailed information, see the OpenMP Web site [4].

Portability is an important characteristic of OpenMP. Any compiler that supports OpenMP can be used to compile parallel application source code that is developed using OpenMP. The resulting compiled binary should be run on the target hardware platform to achieve parallel performance.

Simple examples of OpenMP programs written in both C/C++ and Fortran are shown in Figures 3a and 3b. In these examples, the loop iterations that add the y array to the x array are executed in parallel. OpenMP constructs are expressed by pragmas, directives, and programming API calls in the source code, and allow programmers to specify parallel regions, synchronization, and data scope attributes. OpenMP also supports the use of run-time environment variables to specify the run-time configuration. For example, the environment variable OMP_NUM_THREADS specifies the number of working threads used at run-time.

```
#pragma omp parallel default (none) \
shared(n, x, y) private (i)
{
#pragma omp for
    for ( i = 0; i < n; i++)
        x[i] += y[i];
}  /*-- End of Parallel region -- */
```

Figure 3a. An OpenMP example program in C/C++

```
!$omp parallel default (none)
!$omp shared(n,x,y) private(i)
!$omp do
      do i = 1, n
           x(i) = x(i) + y(i)
      end do
!$ end do
!$ end parallel
```

Figure 3b. An OpenMP example program in Fortran

Dynamic Tasking in OpenMP 3.0

OpenMP 3.0 enhances OpenMP by adding the concept of task as specified by the following pragma.

```
#pragma omp task [<clause>][]
 structured block.
```

Figure 4a. An OpenMP example program using the task directive

Tasks are work units that can be executed immediately or deferred until later. Tasks are composed of executable code, and data environment and internal control variables. While a task can be tied to a thread, tasks are usually executed by a group of working threads. A thread encountering a parallel directive creates as many tasks as threads in the group. Tasks can be associated with barriers, so that a thread that encounters a barrier is blocked until the set of associated tasks is completed. A thread blocked by a barrier may move onto other pending tasks. Barriers can be of two varieties, `taskwait` and `taskgroup`. A `taskwait` barrier blocks the encountering task until all child tasks are finished. The example in Figure 4b illustrates the task directive in OpenMP.

```
#pragma omp parallel
{
#pragma omp task    // multiple tasks of xy are created
  xy();
#pragma omp barrier
     // all xy tasks are guaranteed to be complete here
 ..
}
```

Figure 4b. An example of OpenMP task and barrier

The example in Figure 5 further illustrates the `taskwait` concept.

```
int fibonacci(int n) {
  int fib1, fib2;
  if (n < 2) return n;
#pragma omp task shared(fib1)
 fib1 = fib(n-1);
#pragma omp task shared(fib2)
 fib2 = fib(n-2);
#pragma omp taskwait
 return fib1 + fib2;
}
```

Figure 5. An example of OpenMP taskwait

OpenMP Example

The figures below illustrate the usage of OpenMP through the same wordcount program example. The `parser_obj` and `collect_obj` data structures are used as described earlier. The main program using OpenMP directives is shown below in Figures 6a and 6b. It uses the OpenMP task feature to execute a while loop in parallel, which performs the parsing phase and creates the intermediate results. The collect phase is executed in parallel using the `parallel_for` directive. Besides tallying each individual word's count, the OpenMP reduction feature calculates the total number of words along with the number of different words.

```
Int main() {
   …..
  parser_obj* last_parser_obj = new parser_obj();
     while (fgets(line, BUFSIZE-1, input_file_pointer)) {
         copyLinesToParserObject(last_parser_obj);
          parser_obj* old_parser_obj = last_parser_obj ;
           parser_obj* last_parser_obj = new parser_obj();
           last_parser_obj->prev = old_parser_obj;
     }
     parser_obj* parser_list_end = last_parser_obj;
     #pragma omp parallel
     {
      #pragma omp single nowait
      {
       while(last_parser_obj){
        #pragma omp task firstprivate(last_parser_obj)
        {
         last_parser_obj->process();
        }
        last_parser_obj = last_parser_obj->prev;
       }
      }
     }
```

Figure 6a. An example of wordcount using OpenMP — Part 1

```
     last_parser_obj = parser_list_end;

     collect_obj* cobjs[26]; // 26 alphabets, one cobj for every leading alphabet

     int j; int max_cnt = 0; int diff_cnt = 0; int total = 0;
     const char *max_str = NULL;

#pragma omp parallel for reduction (+:total, diff_cnt)
     for (j=0; j < 26; j++) {
      cobjs[j] = new collect_obj(j, last_parser_obj);
      cobjs[j]->process();
      total += cobjs[j]->total_word_count;
      diff_cnt += cobjs[j]->diff_word_count;
     }

#pragma omp parallel for
    for (j=0; j < 26; j++) {
        #pragma omp critical
        if (cobjs[j]->max_word_count > max_cnt){
          {
          max_cnt = cobjs[j]->max_word_count;
```

```
            max_str = cobjs[j]->max_word_str;
            }
           }
         }
} // end of main
```

Figure 6b. An example of wordcount using OpenMP — Part 2

OpenMP Considerations

OpenMP semantics allow the OpenMP pragma to be ignored and run as a sequential program. The biggest benefit of OpenMP is a relatively easy conversion process from sequential legacy program into a parallel OpenMP program. The other big benefit is the ability to eliminate tedious thread creation and synchronization details and allow software developers to focus on core program logic instead of having to manage concurrent threads. Because OpenMP is specified as a shared memory programming model, its parallel scalability ultimately is restricted by the number of processor cores — and thus, threads — available on the target machine. Scalability also is limited by the degree of complexity of the computational logic relative to the internal thread management overhead.

OpenMP 3.0 supports the dynamic task queue construct to make the parallel programming model more flexible. Because OpenMP is language-based and every new construct requires work by the compilers, overall OpenMP functionality is not as rich as the run-time library based models. And finally, data race conditions are the most challenging problem for parallel software developers. While OpenMP can reduce the occurrence of data race conditions due to its higher level of design abstraction, OpenMP provides nothing specific to help detect or avoid data race problems.

Threading Building Block

Intel's TBB [8] is a C++ template library that aims to reduce the amount of effort required to express parallelism in C++ programs. TBB exports a rich set of templates that implement several popular parallel algorithms such as `parallel_for`, `parallel_while`, `parallel_reduce`, and more. TBB also provides parallel data structures such as `concurrent_queue`, `concurrent_vector` and `concurrent_hash` maps that significantly reduce the developer's burden of managing complex and error-prone parallel data structures. TBB has a scalable memory allocator and provides primitives for atomic operations and synchronization. The central execution unit in TBB is a task that gets scheduled by the run-time engine of the library. The underlying task scheduler in TBB is based on the work- stealing scheduler in Cilk [21].

TBB Example

The wordcount example again illustrates the use of several key features in TBB. The previously described `parser_obj` and `collect_obj` classes perform parsing of text files and tallying of intermediate results.

The `parallel_for` breaks up the iteration space into several chunks and allows each thread to work on a chunk. A `blocked_range` is another TBB template that abstracts the one-dimensional iteration space upon which algorithm such as `parallel_for` operates. The class `ApplyParser` (Figure 7a) shows the use of a `parallel_for` template that is invoked from the main program shown in Figure 7c. TBB also allows convenient reduction operations to be performed in parallel. The class `ReduceCollect` (Figure 7b) illustrates the specification of a reduction operation. The `operator()` that works on a blocked range defines the iterative operation for every element in the block. A reduction object requires that a `join()` operation be specified and a constructor with a split argument.

```
using namespace tbb;

class ApplyParser {
    parser_obj **const my_plist;
public:
   void operator() (const blocked_range<size_t>& r) const {
     parser_obj **plist = my_plist;
     for (size_t i=r.begin(); i!=r.end(); ++i)
        plist[i]->process();
   }
   ApplyParser (parser_obj* plist[] ): my_plist(plist){}
};
```

Figure 7a. Wordcount example using TBB — Part 1

```
class ReduceCollect {
   collect_obj **my_clist;
public:
  int my_total_count; int my_diff_count; int my_max_count;
    const char *my_max_str;
    void operator() (const blocked_range<int>& r) {
       collect_obj **clist = my_clist;
        int total_count = my_total_count; int diff_count = my_diff_count;
       int max_count = my_max_count;
       const char *max_str = my_max_str;

       for (int i = r.begin(); i!= end; ++i){
          clist[i]->process();
          total_count += clist[i]->total_word_count;
          diff_count += clist[i]->diff_word_count;
          if(clist[i]->max_word_count > max_count){
             max_count = clist[i]->max_word_count;
             max_str = clist[i]->max_word_str;
          }
        }
        my_total_count = total_count; my_diff_count = diff_count;
        my_max_count = max_count; my_max_str = max_str;
```

```
    }

  ReduceCollect (ReduceCollect& x, split): my_clist(x.my_clist), my_total_count(0),
my_diff_count(0),
                        my_max_count(0), my_max_str(NULL){}
   void join( const ReduceCollect& y) {
        my_total_count += y.my_total_count; my_diff_count += y.my_diff_count;
        if(y.my_max_count > my_max_count) {
                my_max_count = y.my_max_count;
                 my_max_str = y.my_max_str;
        }
   }
   ReduceCollect(collect_obj *clist[] ): my_clist(clist), my_total_count(0),
my_diff_count(0),                          my_max_count(0), my_max_str(NULL){}
};
```

Figure 7b. Wordcount example using TBB — Part 2

```
void ParallelReduceCollect (collect_obj *clist[], int n) {
     ReduceCollect rc (clist);
     parallel_reduce(blocked_range<int>(0,n), rc);
     total = rc.my_total_count;
     diff_cnt = rc.my_diff_count;
     max_cnt = rc.my_max_count;
     max_str = rc.my_max_str;
}
int main(int argc, char* argv[]) {
        task_scheduler_init init;
        parser_obj* pobj = new parser_obj();
        int size = 0;
        while (fgets(line, BUFSIZE-1, input_file_pointer)) {
                copyLinesToParserObjects();
                size++;
                parser_obj* old_pobj = pobj ;
                parser_obj* pobj = new parser_obj();
                pobj->prev = old_pobj;
        }

        parser_obj *ptmp=pobj;
        parser_obj **pList;
        pList = new parser_obj*[size];

        for(int i=0; i < size; i++) { pList[i] = ptmp; ptmp = ptmp->prev; }

        parallel_for(blocked_range<size_t>(0, size), ApplyParser(pList));

        collect_obj* cobjs[26]; // 26 alphabets, one cobj for every leading
alphabet
```

```
        for (int j=0; j < 26; j++) {
           cobjs[j] = new collect_obj(j, pobj);
        ParallelReduceCollect(cobjs, 26);
}
```

Figure 7b. Wordcount example using TBB — Part 2

TBB Considerations

TBB is the best-known shared memory parallel programming model based on run-time libraries today. With TBB, templates containing a rich set of common parallel constructs and algorithms significantly reduce the effort required to write parallel programs. The TBB approach also shields software developers from nuances of thread management, and the risk of introducing programming errors such as data race conditions is greatly reduced due to its higher level of design abstraction. While providing a big step forward in parallel programming productivity, it can still be a very complicated effort to design a non-trivial program with TBB. Furthermore, software developers may feel uncomfortable using the complex C++ templates in TBB. Compared with OpenMP, it is much harder to convert an existing sequential program into a parallel program using TBB, and the TBB program could incur significant overhead when running in a single thread sequential mode.

Parallel Framework Project

The Parallel Framework Project (PFP) [11], formerly known as MCFX, is a new parallel programming framework introduced by Oracle developers at Supercomputing 2009. Currently in its infancy, PFP introduces the concept of customizable domain specific schedulers — or executable containers — at the application level. The genesis of PFP stems from continued parallel programming challenges despite higher level constructs proposed by the industry and academia in recent years. Although some lower level details are abstracted, general-purpose parallel programming still forces developers to think in terms of splitting algorithms into tasks for scheduling on multiple cores or processors, shifting the focus from program data structures to the tedious work of partitioning and scheduling.

Programmers can benefit from a customized set of domain specific reusable parallel objects built by domain experts with basic knowledge of PFP. The PFP concept of executable objects can be used to model real-world entities or objects in application domains. When an executable object is placed in an executable container, the associated process of the object is executed concurrently with the processes of other executable objects in the host executable container. The behavior of an executable object can depend on its internal states or external conditions, making it possible to model dependencies among various application objects. An executable object can generate more executable objects that can be added back to the executable container.

PFP Example

The use of PFP is illustrated using the previously described wordcount example. The `collect_obj and parser_obj` classes are used as described in the OpenMP example earlier. Here only the differences specific to PFP are outlined. The `parser_obj` and `collect_obj` are derived from the PFP base `class ex_object_t` (Figure 8a). The main program shows how a PFP executable container is instantiated and invoked (Figure 8b).

```
class parser_obj : public ex_object_t {
 ...
 virtual int process(executable_container_t* queue);
};
int parser_obj::process(executable_container_t* a_queue) {
 ...
  return EC_NORMAL;
}
class collect_obj : public ex_object_t {
 ...
 virtual int process(executable_container_t* queue);
 static unsigned int diff_red_id;
 static unsigned int total_red_id;
 static unsigned int max_red_id;
};
int collect_obj::process(executable_container_t* w_queue) {
 ...
  w_queue->reduce( this, this->diff_red_id, &diff_word_count);
  w_queue->reduce( this, this->total_red_id, &total_word_count);
  w_queue->reduce( this, this->max_red_id, &max_word_count);
  return EC_NORMAL;
}
```

Figure 8a. Wordcount example using PFP — Part 1

```
int main() {

    ec_queue_t parser_list(nth, ECP_NO_CHECK);

         parser_obj* pobj = new parser_obj();
    while (fgets(line, BUFSIZE-1, input_file_pointer)) {
        copyLinesToParserObjects();
        parser_obj* old_pobj = pobj ;
        parser_obj* pobj = new parser_obj();
        pobj->prev = old_pobj;
        parser_list.push(old_pobj);
    }

    parser_list.run();

    sum_reducer_t<int> diff_reducer;
         sum_reducer_t<int> total_reducer;
    max_reducer_t<int> max_reducer;
```

```
    ec_queue_t collect_list(nth, ECP_NO_CHECK);

        collect_obj* cobjs[26]; // 26 alphabets, one cobj for every leading
alphabet
    for (int j = 0; j < 26; j++) {
      cobjs[j] = new collect_obj(j, pobj);
      collect_list.push( cobjs[j] );
    }
    collect_list.run();
}
```

Figure 8b. Wordcount example using PFP — Part 2

PFP Considerations

The PFP framework is implemented through a C++ class library with a thin run-time layer. PFP takes the idea of high-level abstraction for parallelism as found in Intel's TBB and takes it a step further by providing an even higher-level abstraction for a few specific domains. PFP also encourages the reuse of the existing parallel objects with its object-oriented programming model. While the implementation of PFP itself is in its early stages, this approach holds considerable promise as it shields novice programmers from having to think in parallel while providing the benefits of multicore system parallel performance.

Many experts in industry and academia have reached the conclusion that there is no silver bullet to solve all aspects of parallel programming. For this reason, domain-specific experts unfamiliar with parallel programming often deem general-purpose frameworks inadequate. PFP aims to provide a productive parallel software environment for supporters of targeted domains. However, PFP suffers from the same drawback as TBB in that — unlike the OpenMP model — it is relatively difficult to parallelize existing legacy programs. In addition, due to the narrow focus of a few specific domains, PFP may not be appropriate to tackle problems in non-targeted domains.

MapReduce and Hadoop

MapReduce [12] is a software framework for processing large data sets in a distributed manner. A user specifies a map function to process a large volume of source data and generate a set of intermediate representations with key-values in parallel. A reduce function then merges all intermediate values associated with the same intermediate key. This programming model, along with a proprietary implementation, was developed at Google. Many real-world problems can be expressed using the MapReduce model — distributed `grep`, count of URL access, reverse Web-link graphs, inverted index, and more. Although the programming model does not necessarily assume a shared or distributed memory implementation, many popular MapReduce implementations work on terabytes of data on large clusters of commodity machines, thus adopting a distributed model. The run-time system takes care of the details of partitioning input data, scheduling the program's execution across machines, handling machine failures, and managing inter-machine communication. As a result, users can concentrate on expressing the MapReduce semantics without worrying about parallel programming or message passing semantics.

Apache's Hadoop [13] is a popular open-source framework that implements the MapReduce computational paradigm. In addition to the ability to express MapReduce functionality, the Hadoop run-time system consists of the Hadoop Distributed File System (HDFS), which achieves reliability by replicating data across multiple hosts. The heart of Hadoop run-time is the central job tracker to which client applications submit MapReduce jobs. The job-

tracker in turn distributes the jobs to task-trackers running on the cluster trying to keep data as close to computation as possible.

Hadoop Example

The wordcount example in Figures 9a and 9b, adapted from the Hadoop Tutorial [13], is written in Java™ and illustrates how Hadoop performs MapReduce.

```
public class WordCount {

    public static class Map extends MapReduceBase implements
                    Mapper<LongWritable, Text, Text, IntWritable> {

      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map (LongWritable key, Text value,
         OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException {

            String line = value.toString();

            StringTokenizer tokenizer = new StringTokenizer(line);

            while (tokenizer.hasMoreTokens()) {
               word.set(tokenizer.nextToken());
               output.collect(word, one);
            }
       }
     }
```

Figure 9a. Wordcount example using Hadoop — Part 1

```
    public static class Reduce extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
  }
}
```

Figure 9b. Wordcount example using Hadoop — Part 2

Hadoop Considerations

The Hadoop programming and run-time model is well suited to cloud computing. Indeed, an increasingly popular solution for commodity cloud platforms, such as Amazon Elastic Compute Cloud (Amazon EC2) [20], is to run Hadoop in conjunction with Amazon Simple Storage Service (Amazon S3). Although remote cloud storage platforms negate some of the obvious locality benefits of data proximity, ease of use is a strong factor driving the adoption of Hadoop in cloud computing. In addition, Hadoop is fast becoming the chosen technology to implement compute-intensive tasks on grid engines and clusters. The Hadoop framework itself is implemented in Java, but Hadoop tasks can be written in any other programming language. As of this writing, the adoption of Hadoop seems to be mostly limited to Java and a few other dynamic languages.

Message Passing Interface

The Message Passing Interface (MPI) [6, 7] is an industry-standard API specification designed for high-performance computing on multiprocessor machines and clusters. The standard was designed jointly by a broad group of computer vendors and software developers, with a number of MPI implementations produced by different research institutes and companies. The most popular one is MPICH, which often is used to optimize MPI implementations for a specific platform or interconnect [6].

MPI offers a distributed memory programming model for parallel applications. Although the entire MPI API set contains more than 300 routines, many MPI applications can be programmed with less than a dozen basic routines. Figure 10 shows a pair of send and receive routines to communicate a message. The first three arguments of both routines specify the location, data type, and size of the message, and the fourth argument identifies the target process with which to communicate. The fifth argument provides a further mechanism to distinguish between different messages and the sixth argument specifies the communication context. The receive routine contains an additional argument to report the message reception status.

```
MPI_Send (
   void* message_buffer,            // which data
   int count,                       // data size
 MPI_Datatype datatype,             // data type
 int destination,                   // destination MPI process
 int  message_tag,                  // message ID tag
 MPI_Comm communicator);            // communication context


MPI_Recv (void* message_buffer,     // which data
   int count,                       // data size
 MPI_Datatype datatype,             // data type
 int source,                        // source MPI process
 int  message_tag,                  // message ID tag
 MPI_Comm communicator,             // communication context
 MPI_Status* status);               // message status
```

Figure 10. MPI Send and Receive

In MPICH design, the entire API set is implemented and built on top of a small core of low-level device interconnect routines. This design feature supports portability across different platforms — a developer only need re-work the core set of device interconnect routines to port or optimize MPICH for a new platform or interconnect.

The MPI implementations are still evolving. MPI-1 supports key features such as interconnect topology, and point-to-point and collective message communication with a communicator. A message can contain MPI data of primitive or user-defined (derived) data types with message data content in packed or unpacked format. MPI-2 provides many advanced communication features such as remote memory access and one-side communication. It also supports dynamic process creation and management, and parallel I/O.

Memory Hierarchy

Given the current state of semiconductor technology and computer architecture, the dominant system performance factor is memory hierarchy rather than CPU clock rate. Figure 11 shows a graph of non- uniform memory

performance. An application runs faster if most of its instruction and data memory accesses fall within the cache range. Being a distributed memory programming model, MPI usually can achieve good linear scalability for large-scale applications. When an MPI application is partitioned to run on a large cluster of computing nodes, the memory space of each MPI process is reduced and the memory accesses could fall outside the high performance range of the memory hierarchy. This non- uniform memory performance effect can also apply to the other programming models discussed in this paper.

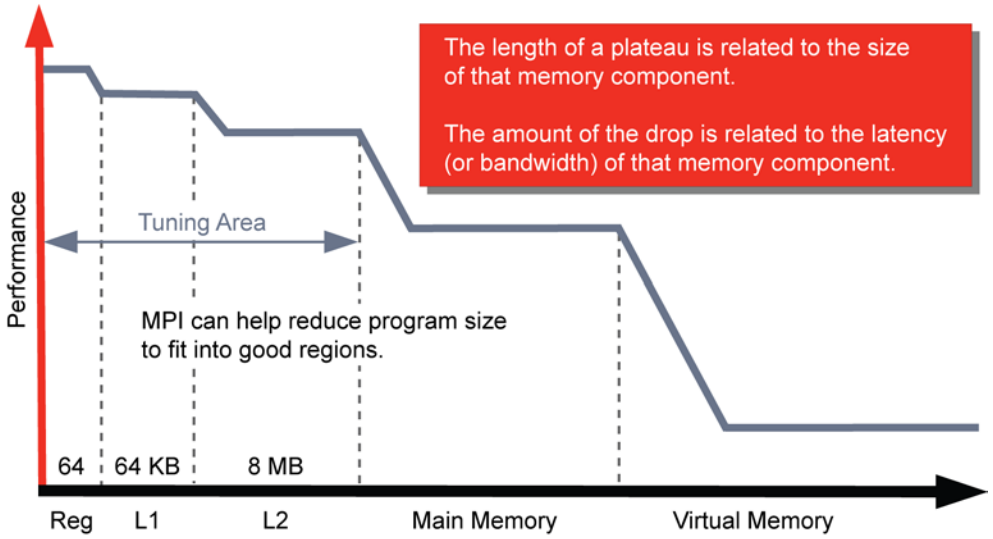

Figure 11. Non-Uniform Memory Performance

```
#include <mpi.h>

int max_cnt;
int
total_cnt;
int diff_cnt;
const char *max_str = NULL;
```

```
struct StrLess
{
   bool operator() (const char* s1, const char* s2) const
     { return strcmp(s1, s2) < 0;
   }
};
typedef std::map<const char*, int, StrLess> word_map_t;

static void parseAndCollect(
         char line[], word_map_t& wmap, int alpha_index_begin, int alpha_index_end)
{
  int idx = 0;
  char word_buf[124];
  char lead_char_range_begin = 'a' +
  alpha_index_begin; char lead_char_range_end = 'a' +
  alpha_index_end;

  while (line[idx] != '\0') {
    while ( isalnum(line[idx]) == 0) { // skip non alnum
      chars idx++; if (line[idx] == '\0') return;
   }
   int cnt = 0;
   while( isalnum(line[idx]) != 0) {
        if (line[idx] == '\0')
        break;
        // convert the word to lower case
        word_buf[cnt] = ((isupper(line[idx])!=0) ?
        tolower(line[idx]):line[idx]); idx++; cnt++;
   }
   word_buf[cnt] = '\0';
   if (word_buf[0] >= lead_char_range_begin && word_buf[0] <=
      lead_char_range_end){ char* new_word = new char[strlen(word_buf) + 1];
      strcpy(new_word,
      word_buf); char* cur_word
      = new_word;

      word_map_t::iterator hit =
      wmap.find(cur_word); total_cnt++;
      if (hit != wmap.end()) {
           (*hit).second = (*hit).second + 1; // increase the count.
           if ((*hit).second > max_cnt) {max_cnt=(*hit).second; max_str =(*hit).first; }
      }
      else {
         word_map_t::value_type new_item(cur_word,
         1); wmap.insert( new_item );
          if (max_cnt < 1) { max_cnt = 1; max_str = cur_word;
          } diff_cnt++;
      }
```

Figure 12a. Wordcount example using MPI — Part 1

```
int main(int argc, char* argv[]) {

    int rank, size, alpha_index_begin, alpha_index_end;
    MPI_Status mpi_status;
    MPI_Init(&argc,&argv);                    /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &size);  /* get number of processes */

    max cnt = 0; total cnt = 0; diff cnt = 0;

    int portion = 26 / size; if (26 % size) portion++;

    alpha_index_begin = rank* portion;
    alpha_index_end = alpha_index_begin + (portion -1);
    if (alpha index end > 25) alpha index end = 25;

    FILE* fp = fopen( “input.txt”, "rt");

    word map t wmap;

    char line[BUFSIZE];
    int line count = 0; diff cnt = 0;

    while (fgets(line, BUFSIZE-1, fp)) {
      if (line[strlen(line) - 1] == '\n')    line[strlen(line)-1] = '\0';
      char* new_line = new char[strlen(line) + 1];
      strcpy(new_line, line);
      parseAndCollect(new_line, wmap, alpha_index_begin, alpha_index_end);
      line count++;
    }

    fclose( fp );

    int final total cnt, final diff cnt, final max cnt;

   MPI_Reduce(&diff_cnt, &final_diff_cnt, 1, MPI_INT, MPI_SUM, 0,MPI_COMM_WORLD);
   MPI_Reduce(&total_cnt, &final_total_cnt, 1, MPI_INT, MPI_SUM, 0,MPI_COMM_WORLD);
   MPI_Reduce(&max_cnt, &final_max_cnt, 1, MPI_INT, MPI_MAX, 0,MPI_COMM_WORLD);

        if (rank == 0)
     fprintf(stderr,"Total words %d Diff cnt %d Max Cnt %d\n",
              final total cnt, final diff cnt, final max cnt);

    MPI_Finalize();
    return 0;
}
```

Figure 12b. Wordcount example using MPI — Part 2

MPI Example

Figures 12a and 12b illustrate the usage of MPI with the previously described wordcount example. Here, due to the communication cost involved in passing messages between processes, the implemented algorithm is significantly different. Multiple processes read the input file with each process responsible for counting words starting with a range of designated letters of the alphabet, for example from 'a' to 'l'.

MPI Considerations

MPI offers a good solution for parallel application on computer clusters, but it can be a difficult programming model for many developers. Because MPI has longer communication latency, the program core logic must be partitioned well to justify the distribution overhead. Analyzing, partitioning, and mapping an application problem to a set of distributed processes is not an intuitive task. Because of the interactive complexity of many MPI processes, it is also quite challenging to debug and tune a MPI application running on a large number of nodes, even with the right tools. The implementation quality of MPI routines can impose additional software development challenges. MPI performance depends on the underlying hardware platform and interconnect — in some extreme cases, the MPI application behavior may be affected by heavy interconnect traffic. Another big issue is the reliability of large-scale MPI applications. Depending on the implementation, an MPI program could stop working whenever a single computing node fails to respond correctly.

Cloud and Grid Computing

Grid computing [18] virtualizes compute resources with the goal of delivering them as a utility service. Cloud computing is a hybrid of grid computing and SOA [3]. The term cloud computing refers to an application delivered as a service over the internet, computing software and hardware in the data center, or specific tools or infrastructure delivered within a large business enterprise network. Cloud computing can manifest itself in a variety of forms such as Software-as-a-Service, Platform-as-a- Service, Tools-as-a-Service, and Infrastructure-as-a-Service. Clouds can be accessible to the general public or deployed privately, to be used behind an organization firewall.

Modern Web services technology facilitates easy access, and virtualized computing environments support platform portability, contributing to the rise in popularity of cloud computing. But as Berkeley researchers pointed out, cloud computing offers three new aspects from a hardware point of view: [19]

1. The delivery of infinite computing resources available on demand
2. The elimination of up-front commitments by cloud users
3. The ability to pay for use of computing resources on a short-term basis as needed

Because cloud computing permits access to large-scale computing resources from anywhere at any time, it opens the door to possibilities for parallel computing. Provisioning a massive amount of compute resources from a cloud computing utility vendor such as Amazon's EC2 [20] is easy to do. Application service providers offer enhancements that allow users to configure virtualized resources for special scenarios. While cloud computing addresses the utility usage of compute resources, it remains a big challenge for application service providers to adopt parallel application frameworks so that developers can harness the resources made available by cloud computing.

Grid Computing and the Distributed Resource Manager

Grid computing is a form of distributed computing with a utility computing delivery mechanism. The grid computing model tends to be loosely coupled and heterogeneous. At the heart of grid computing is the Distributed Resource Manager (DRM), containing a master daemon to schedule and dispatch submitted tasks in the job queue(s). The master daemon constantly monitors the status of the execution nodes in the grid in order to dispatch a task to the most appropriate node. An administration module manages the system configuration, accounting, and resource

usage policy. Currently the leading DRM market leader is the Load Sharing Facility (LSF) from Platform Computing, but it requires an expensive license fee.

Oracle's fully integrated DRM, Oracle Grid Engine, runs on some of the most popular operating systems such as Oracle Solaris, Windows, Linux, HP-UX, IBM AIX, and Apple Macintosh OS/X. It also supports the standard DRM Application API DRMAA for Java and native language binding.

Cloud Computing Considerations

While cloud computing is gaining broad acceptance, there are still no widely adopted software tools and infrastructure to enable a robust parallel software environment for cloud computing. Amazon Elastic Compute Cloud allows application developers to build and port applications on the cloud and provision compute resources with some flexibility. However, Amazon does not offer a productive and efficient, scalable high abstraction software development framework. The Google App Engine [21] allows application developers to build Web applications on the Google cloud. Although Google App Engine offers Java and Python run-time environments with APIs and task queue mechanisms, the API design patterns are limited. In comparison with Amazon EC2, the Google App Engine environment is less flexible and configurable. Overall, cloud computing puts more focus on Web applications with computing and storage resources provisioning than user scalability today. However the need for parallel application performance scalability still remains unsolved.

Grand Central Dispatch

Apple's Grand Central Dispatch (GCD) [14] offers a novel approach that allows applications to take advantage of multicore system parallel processing power without requiring tedious thread management and synchronization in the application software. Its basic design supports the application-level task queue mechanism at the operating system level. This approach supports the scheduling of multiple tasks in a program to execute in parallel on any available processor core.

The key elements of GCD's programming model are blocks and queues. Tasks in GCD can be manifested as a function or a block. Apple also introduced C, C++, and Objective C extensions to support task blocks. A block syntax is quite similar to C++ new lambda extension. Figure 13 shows a simple block example.

```
x = ^{printf{"hello world\n"); }
```

Figure 13. Block extension code sample

Variable x can be invoked just like a function as in x(). Blocks can be scheduled for execution by placing them on system-level or user-level queues. GCD also supports the concept of event source to help achieve synchronization of multiple blocks.

GCD Considerations

There are several benefits to GCD. The first obvious benefit is the reduction of programming complexity in creating and managing concurrent threads. The second benefit is that blocks facilitate the ease of scheduling small chunks of work to the global queue for parallel execution. For these reasons, an application designed with GCD can achieve better parallel scalability by partitioning the critical code into finer grained parallel tasks. The most significant benefit of GCD is support for task scheduling at the operating system level, which achieves better processor resource utilization of the entire system.

However, in a tightly-coupled parallel application that is latency-driven instead of throughput-oriented, the task queue at the system level could contribute unnecessary overhead.

Real Case Study

The following sections examine two classes of real-world applications to evaluate the design challenges of parallel application development.

Parallel Search

Tree search is a popular approach for finding the optimal solution or enumerating all the possible solutions of a huge data space in solving well-formulated problems. Many scientific or business problems can be transformed and formulated as parallel tree searches, and the application of parallel tree searches can range from trivial puzzle solving to complex strategic analysis. One common approach is to construct a state tree with each node representing a specific application state, and search through the state tree to find the target node(s) with the optimal state or to reach a target node in a minimum number of steps.

It takes two steps to develop software for a parallel search application. The first step is to create a good abstraction model to transform the target problem and represent it in a state tree. The quality of the abstraction model affects the tree shape and tree size, which then affects the required searching

effort — in general, the finer the model, the bigger the state tree needed. The first step tends to be a difficult modeling task and requires good domain expertise to construct a good quality tree for even a modestly complex problem. The second step is more difficult than it looks and involves searching through the state tree using parallel computing. In most cases, the tree is dynamically constructed during the program run time. Because the tree tends to grow in an unbalanced way, it is easy to lose load balance by assigning a child branch randomly to a working thread or process. A more suitable approach is to schedule a node operation in a task queue and distribute the tasks to the working threads or processes. However, even this approach does not work when a tree node operation is relatively lightweight compared to the queuing overhead.

In addition, communication overhead is another big problem. A working thread needs to communicate to the other threads what nodes it has already processed in order to prevent duplicated effort by the threads in processing the same or similar nodes. When the working thread encounters a solution node it needs to communicate with the global data structure to check if the newly found solution node is an optimal node and interacts with the new node accordingly.

The challenges of designing a parallel search program start with the model abstraction, continue with tree construction, and end with parallel search algorithms. A good tree model needs domain-specific expertise to formulate the problem and represent it with an efficient tree node presentation. After the state tree is constructed, it usually requires experimental analysis on the tree patterns to design the right searching algorithm. A well-designed algorithm should be able to minimize the thread dispatching and communication overhead to achieve good parallel scalability.

Parallel Simulation

Simulations are vital applications in various industries. The financial industry employs Monte Carlo simulations, the semiconductor industry uses Electronic Design Automation (EDA) logic simulations, life sciences utilize protein folding simulations, and more. The biggest challenge of large-scale simulations is dealing with vast amounts of objects and data. Many applications like EDA logic simulations or life science molecular simulations often handle millions or even billions of modeling objects. It is unrealistic to create millions of threads or processes to model simulation objects with threads on a one-to-one basis. Indeed, it takes extensive computations to evaluate the

interaction among neighboring objects and propagate the impact of that interaction to other objects until the overall effect dies down.

Discrete type simulations usually proceed as a cycle-based or event-driven simulation. Cycle-based modeling tends to be rigid and less accurate, however computation is straightforward and easier to parallelize. If the computing tasks in a given cycle can be sorted according to data dependency, the parallel simulation can be simplified to a `parallel_for loop` iterating through the ordered tasks with a dependency constraint for each simulation cycle.

The event-driven simulation modeling is flexible and more accurate in nature, however parallelization efforts can be quite challenging. At any given time tick, the event object evaluation task and newly created event object scheduling task must be parallelized. Because a synchronization barrier usually is imposed on the time boundary, simulation parallel scalability is restricted and not uniform through the time ticks.

When the conventional parallel programming method is inadequate to handle a large-scale simulation, the solution sacrifices modeling accuracy by taking a mixed-level modeling approach to work around extensive computation and communication problems. The compromised approach applies more coarsely grained modeling in the higher hierarchical level to reduce the amount of objects and approximate the interaction effect among neighboring objects. Finer grained modeling can be applied to the hot spot regions for more accurate simulation. For some applications, simulation developers may even take the extra effort to build an application-specific hardware accelerator to solve the problem.

Developer Tools

Many programmers think Oracle Solaris Studio [23] provides the best integrated set of developer tools for parallel application development. This section focuses discusses how developers can leverage the tools in Oracle Solaris Studio to develop parallel applications throughout the four main stages in the software development cycle — compilation, debugging, performance tuning, and thread analysis.

Compilation

The most fundamental tool in the application development life cycle is a compiler. The quality of a compiler determines the ultimate performance and condition of executable code. The compiler should support the latest OpenMP standard and other popular parallel run-time libraries. In addition, the compiler's internal working thread implementation profoundly determines the application's parallel performance when running on chip multithreading (CMT) servers or chip multiprocessor (CMP) machines. Oracle Solaris Studio compilers have consistently produced many SPEC® OpenMP Benchmark Suite (SPEC OMP)[1] world records on Oracle symmetrical multiprocessing systems in recent years.

A few compilers, such as the Oracle Solaris Studio and Intel compilers, can analyze and parallelize a sequential application into a multithreaded application automatically. In general, it is very difficult to perform automatic parallelization on large sections of a program. For that reason, the compilers tend to focus on loop constructs. Oracle Solaris Studio compilers analyze the application data dependency of loop iterations and parallelize the iterations into multiple threads if there are no data dependencies.

Debugging

Debugging is a very time consuming stage in the application development cycle. A good debugging environment should consist of a fully integrated debugging GUI and a powerful debugging back-end such as dbx. Oracle Solaris

1 SPEC is a registered trademark of the Standard Performance Evaluation Corporation (SPEC).

Studio compilers produce better quality symbol mapping data for debugging than most compilers, resulting in better transparency when dealing with program data and instructions.

The fully integrated GUI environment in Oracle Solaris Studio allows developers to set and manage breakpoints and watchpoints effectively. It provides calling stack and local variable windows to display the current program context at any time. It is very easy to navigate the program flow by applying single step and step-in commands in the source code editor window directly. The fix and continue feature can cache the incremental fixes to shorten build times and speed the debugging cycle.

The Oracle Solaris Studio debugging environment supports the above features for OpenMP programs and POSIX thread (Pthread) applications. Additionally, Oracle Solaris Studio provides a multisession debugging feature that can load multiple program instances and debug them simultaneously.

Developers can use this feature to debug and compare a faulty multithreaded version with a reference sequential version of the same application program.

Performance Tuning

The performance analyzer in Oracle Solaris Studio is one of the best performance tools in the industry. The tool supports the C, C++, Fortran, and Java programming languages. Oracle Solaris Studio performance analyzer is easy to use and works with unmodified binaries. Developers need only to recompile the source codes with the `-g` option (or `-g0` for C++) to get the complete source line level information. It works on parallel programs designed by automatic parallelization or explicit parallelization methods such as OpenMP, MPI, and Pthreads.

It takes two simple steps to use the performance analyzer. First, the performance analyzer collects the profiling data by running experiment(s). Data is collected by applying statistical callstack samplings triggered by clock or hardware counters. Next, the performance analyzer loads the performance data and presents them in multiple presentation views. Developers can navigate and examine the performance data at the routine, statement, and instruction levels. Developers can collect and print the performance data in batch command mode or interactive mode.

The performance analyzer can load multiple experiments to aggregate and filter the data by experiment, thread, function, and time. The performance analyzer also provides a set of API routines to allow developers to instrument codes manually. The Oracle Solaris Studio Performance Analyzer supports Java profiling in three modes. In user mode it only shows the user threads and Java callstacks. In expert mode it shows all threads and Java callstacks for user threads and machine callstacks for other threads. In machine mode it shows all threads and machine callstacks for all threads.

Thread Analysis

Arguably the single most difficult problems faced by programmers writing parallel applications are data race conditions and the closely related deadlocks. In spite of the availability of a wide variety of synchronization primitives, it is very hard to avoid the occurrence of data races in most real-world applications with any degree of complexity. Therefore, it is imperative that any software programming framework for expressing parallelism is complemented by a race detection tool. Data races occur when two or more threads access the same memory location concurrently, at least one of the accesses is attempting to write to that location, and the threads are not using any exclusive locks to control the access to that memory location. Because the order of these memory accesses is non-deterministic, the computation can give different results from run to run. While some data races are benign, most data races are actual bugs in the parallel program and must be detected and fixed to ensure program correctness.

A deadlock is a condition where two or more threads are blocked forever because they are waiting for each other to release a shared resource. A good example of such a shared resource is a mutual exclusion lock. A typical scenario

is a thread holding a lock and requesting another lock while a second thread holds the requested lock and is waiting for the first thread to relinquish the first lock before releasing its own.

Oracle Solaris Studio has a state-of-the-art thread analysis tool [24] that is very effective in finding data races. It works on multithreaded programs written using POSIX, Oracle Solaris threads, or OpenMP. The thread analyzer requires the application code to be instrumented using an explicit compile time option. At run-time the thread analyzer traces memory operations and thread management and synchronization operations. It eventually compares events for two threads to detect a possible data race. The virtual addresses of potential data races are highlighted and linked back to the source code using a graphical user interface. Occasionally the thread analyzer reports false positives — data races that have not actually occurred in the program. False positives often are caused by user-defined synchronization mechanisms that are not recognized by the tool.

Summary

With so many emerging software technologies to choose from, it is quite challenging for software professionals to decide which parallel programming framework and tools to adopt for developing an application. The most important thing, however, remains fundamental software development skill and experience. Beginning with problem definition, developers need to analyze, abstract, and map the problem into a data process flow. The data structure and algorithm must be designed for parallel execution from the outset. Expending the extra effort early in the design phase might be a big change from usual practice, but it is a small price to pay compared to the reward the developer gains from improved parallel performance and better software quality.

References

[1] Qusay H. Mahmoud *Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)* http://www.oracle.com/technetwork/articles/javase/soa-142870.html

[2] *The Open Group — SOA Work Group Members Site* http://www.opengroup.org/projects/soa/

[3] *NIST Definition of Cloud Computing v15* http://www.nist.gov/itl/cloud/

[4] *The OpenMP API specification for Parallel Programming* http://www.openmp.org

[5] Barbara Chapman, Gabriele Jost and Ruud Van Der Paas, *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press 2007.

[6] *The Message Passing Interface Standard*. http://www.mcs.anl.gov/research/projects/mpi/

[7] *MPI: A Message Passing Interface Standard Version 2.2* http://www.mpi-forum.org/docs/mpi- 2.2/mpi22-report.pdf

[8] *Intel's Threading Building Blocks Tutorial* https://www.threadingbuildingblocks.org/documentation

[9] Daan Leijen, Wolfram Schulte, and Sebastian Burkhardt *The Design of a Task Parallel Library* Proceedings of the 24th ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. 2009.

[10] Kenny Kerr*, Visual C++ 2010 And The Parallel Patterns Library*. MSDN Magazine, Issue 2009 February

[11] Liang Chen, Deepankar Bairagi and Yuan Lin. *MCFX: A New Parallel Programming Framework for multicore systems.* Proceedings of International Supercomputing Conference. 2009. http://www.springerlink.com/content/9004q3172421j810/

[12] Jeff Dean and Sanjay Ghemawat. MapReduce: *Simplified Data Processing on Large Clusters* https://www.threadingbuildingbhttp://research.google.com/archive/mapreduce.htmlocks.org/documentation

[13] *The Hadoop Tutorial* https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[14] *Grand Central Dispatch: a better way to do multicore.* https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/

[15] *OpenCL - The open standard for parallel programming of heterogeneous systems* http://www.khronos.org/opencl/

[16] *Berkeley UPC User's Guide version 2.10.2* http://upc.lbl.gov/docs/user/index.shtml

[17] *CUDA Zone* http://www.nvidia.com/object/cuda_home_new.html

[18] *Oracle Grid Engine* http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

[19] Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia *Above the Clouds: A Berkeley View of Cloud Computing http://berkeleyclouds.blogspot.com/*

[20] Amazon Elastic Compute Cloud (Amazon EC2) http://aws.amazon.com/ec2/

[21] *Intel Cilk++ Software Development Kit* http://software.intel.com/en-us/articles/intel-cilk/

[22] *Google App Engine* http://code.google.com/appengine/

[23] *Oracle Solaris Studio* http://www.oracle.com/technetwork/server- storage/solarisstudio/overview/index.html

[24] *Oracle Solaris Studio 12.2: Thread Analyzer User's Guide* http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html

Acknowledgments

This white paper evolved from an earlier article written by Liang Chen, Develop Applications for Parallel Computing, and posted on the Sun Developer Network Website. Former Sun colleagues Mike Ball and Yuan Lin made significant contributions to that article. Ruud van der Pas at Oracle also contributed important content to that article that has been reused in this white paper. Finally we want to thank our colleagues Vijay Tatkar, Kuldip Oberoi, Don Krestch, and Bhawna Mittal at Oracle for their excellent input and support.

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

ORACLE

CONNECT WITH US

blogs.oracle.com/oracle
facebook.com/oracle
twitter.com/oracle
oracle.com

Integrated Cloud Applications & Platform Services

Developing Parallel Programs - A Discussion of Popular Models
May 2016
Author: Liang T. Chen, Deepankar Bairagi