Modules for Teaching Parallel Performance Concepts

Apan Qasem

Abstract This chapter introduces three teaching modules centered on parallel performance concepts. Performance related topics embody many fundamental ideas in parallel computing. In the ACM/IEEE curricular guidelines (ACM2013), an entire knowledge unit has been devoted to parallel performance. In addition, performance topics pervade every knowledge area within PDC and can be found across other knowledge areas including Algorithms, Architecture and Systems Fundamentals. The three modules presented in this chapter cover a range of parallel performance topics. Since power savings have become an important consideration from hand-held devices to supercomputers, energy efficiency is also emphasized in each module. The modules focus more on architectural and algorithmic issues rather than the programming aspects. The modules are constructed to illustrate parallel performance issues primarily through code examples and experimental studies. This approach makes the modules accessible to students who do not *yet* have a strong background in parallel programming. Thus, the target audience for this chapter are instructors who are teaching CS1, with or without parallel programming, and also instructors who are teaching upper-level electives where their students may already have taken a semester of parallel programming.

Relevant core courses: CS1, Operating Systems, Computer Architecture

Relevant PDC topics: speedup (C), efficiency (C), Amdahls Law (A), space vs. time (C), power vs. time (C), synchronization and communication (C), task granularity (A), scheduling and mapping on multicore (A), load balancing (A), trade-offs in performance and power (C), Analysis and Evaluation: linear and super linear speedup (C), latency and bandwidth trade-offs, data locality, SMP (C), NUMA (C), strong and weak scaling (C), (Bloom classification in parentheses)

A. Qasem (✉)
Texas State University, San Marcos, TX, USA
e-mail: apan@txstate.edu

S. K. Prasad et al. (eds.), *Topics in Parallel and Distributed Computing*,
https://doi.org/10.1007/978-3-319-93109-8_3

Context for use: CS1 fundamentals, operating system thread scheduling, parallel architecture performance evaluation

Learning outcomes:

- list and define parallel performance metrics: speedup, efficiency, linear speedup, super linear speedup, latency and bandwidth
- describe the implications of Amdahl's law on parallel performance
- recognize the use of parallelism to achieve strong scaling and weak scaling
- analyze the effects of load imbalances on performance and power
- apply techniques to balance load across threads or processes
- explain the need for inter-thread synchronization and communication
- apply techniques to pin and schedule threads on multicore systems for improved performance
- describe how cores share memory resources, such as DRAM and cache
- recognize the importance of exploiting data locality in parallel applications

Introduction

This chapter introduces three teaching modules centered on parallel performance concepts. Performance related topics embody many fundamental ideas in parallel computing. In the ACM/IEEE 2013 curricular guidelines (ACM2013), an entire knowledge unit has been devoted to parallel performance [1, 2]. In addition, performance topics pervade every knowledge area within PDC and can be found across other knowledge areas including Algorithms, Architecture and Systems Fundamentals.

The three modules presented in this chapter cover a range of parallel performance topics. Since power savings have become an important consideration from hand-held devices to supercomputers, energy efficiency is also emphasized in each module. The topics provide at least 3.5 h of Core-Tier 1, Tier 2 and Elective hours from ACM2013. The modules are designed to be introduced in CS1 and two upper-level electives, namely, Operating Systems and Computer Architecture. They are, however, designed with enough flexibility to enable adoption in a number of undergraduate courses at various levels.

The modules focus more on architectural and algorithmic issues rather than the programming aspects. The modules are constructed to illustrate parallel performance issues primarily through code examples and experimental studies. This approach makes the modules accessible to students who do not *yet* have a strong background in parallel programming. Thus, the target audience for this chapter are instructors who are teaching CS1, with or without parallel programming, and also instructors who are teaching upper-level electives where their students may already have taken a semester of parallel programming.

Elementary Concepts

This module is designed to introduce fundamental concepts in parallel computing in a CS1 course. The concepts are illustrated with no particular binding to any programming language and therefore can be introduced in different flavors of CS1 courses.

Recommended Length 1 lecture (1:15 min)
Recommended Course CS1, CS2

Organization and Content

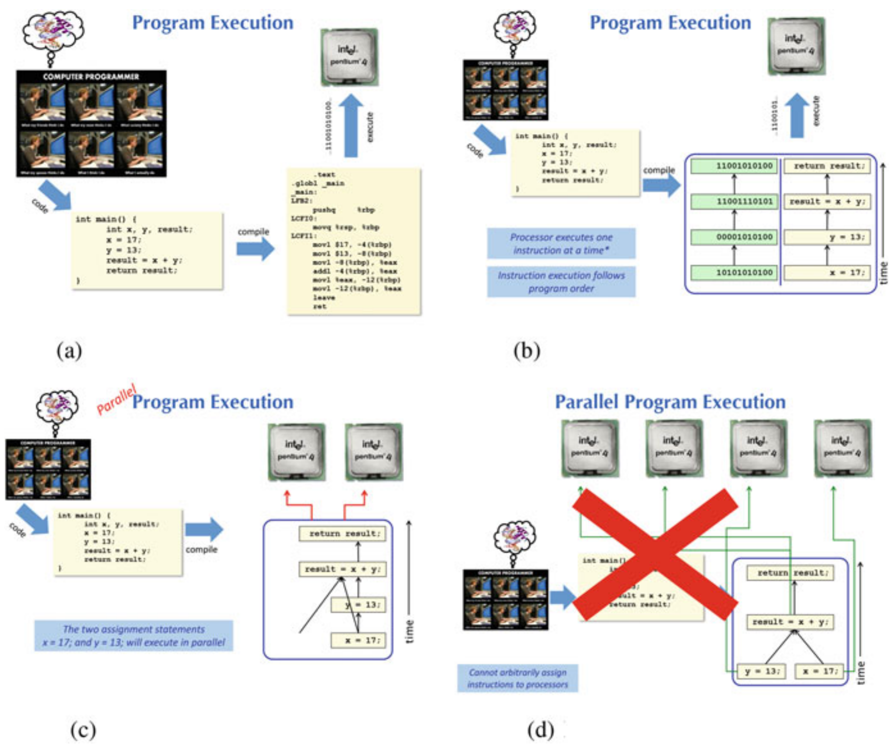
The major topics in this module include (i) overview of parallel computation on a multicore processor, (ii) data dependence and need for synchronization in parallel programs, (iii) parallel performance and Amdahl's law and (iv) energy efficient computing. The topics are introduced through lectures slides, an in-class activity, code examples and a program demo. The following subsections describe how these topics are explained and the order in which they are introduced.

Parallelism in Real Life

The module begins with an in-class activity that engages the students and demonstrates the benefits of parallelism. An activity that works quite well with CS freshman is a live simulation of the word search problem where students act as processing threads. In this activity, the class is split into *k* groups. Each group is assigned the task of finding a collection of words in a book and reporting the page numbers where the words occurred. Each group gets a copy of the book. But the copies are sectioned into different-sized segments. Thus, one group might get the entire book in one chunk while another may be assigned one page per group member. The students are then asked to try to find an efficient method of solving the problem with resources they are given. Naturally, the teams with fewer pages per student (thread) are likely to get to the results first. However, care must be taken in selecting the words and their positions and in segmenting the text.

Parallel Computing and Its Importance Today

Following the in-class example, a set of lecture slides defines parallel computing and discusses its importance in today's world. A high-level definition of a parallel computer is presented. Student familiarity with basic Von-Neumann architecture is assumed (not an unrealistic expectation for CS1 students). The discussion of the definition of a parallel computer is followed by some history of parallel computing.


Program Execution
COMPUTER PROGRAMMER
code
compile
execute
...1100101010...
(a)
Program Execution
code
compile
execute
...1100101...
Processor executes one instruction at a time*
Instruction execution follows program order
11001010100
11001110101
00001010100
10101010100
time
(b)
Parallel Program Execution
code
compile
The two assignment statements x = 17; and y = 13; will execute in parallel
time
(c)
Parallel Program Execution
Cannot arbitrarily assign instructions to processors
time
(d)

```
int main() {
      int x, y, result;
      x = 17;
      y = 13;
      result = x + y;
      return result;
}
```

```
      .text
.globl _main
_main:
LFB2:
      pushq      %rbp
LCFI0:
      movq %rsp, %rbp
LCFI1:
      movl $17, -4(%rbp)
      movl $13, -8(%rbp)
      movl -8(%rbp), %eax
      addl -4(%rbp), %eax
      movl %eax, -12(%rbp)
      movl -12(%rbp), %eax
      leave
      ret
```

```
return result;
result = x + y;
y = 13;
x = 17;
```

Fig. 1 Lecture slides illustrating the differences in serial and parallel program execution. Animation is used for the different blocks in the slideshow

The point is made that parallel computing has been around for a long time, ever since the beginning of computing. Notwithstanding, it has only become mainstream in the last decade. Brief descriptions of mainframe, vector computers and clusters are presented. This is followed by a discussion of multicore computers of today. The importance of energy efficiency and the role it has played in the evolution of computer chips and given rise to multicore systems is discussed. The lecture slides emphasize the need for achieving higher performance at lower power consumption or at specified power budgets. The ubiquity of parallel computers is also discussed. Students are asked to guess/comment on the number of processing cores on their smartphones and tablets. Their guesses are then validated against actual numbers. A discussion follows on the need for more parallel processing cores.

Sequential vs. Parallel Program Execution

A major portion of the module is spent introducing the student to the fundamental difference in sequential and parallel program execution. A walk-through example

Fig. 2 A simple parallel code written in SimPar

```
int add() {
  int x, y, result;
  #PARALLEL {
    x = 17;
    y = 13;
  }
  result = x + y;
  return result;
}
```

Fig. 3 Incorrectly parallelized code

```
int add() {
  int x, y, result;
  #PARALLEL {
    x = 17;
    y = 13;
    result = x + y;
  }
  return result;
}
```

is used for this purpose. Figure 1 shows a subset of the slides that are used to explain this topic. The slides are accompanied by a set of examples written in SimPar [3]. Two such examples are shown in Figs. 2 and 3. SimPar is a simple macro language that uses an intuitive pragma based syntax. Since students are generally not expected to be familiar with any parallel programming language in CS1, SimPar is an effective tool to discuss parallelism with real examples without getting bogged down in syntax minutiae. SimPar contains only one kind of parallel statement, a directive in the form of `#PARALLEL { ...}`. This implies that all high-level statements enclosed in the subsequent block will be executed concurrently. SimPar processes such directives by taking each statement in the block and converting it into a Pthread function. Supplementary materials for this chapter includes a SimPar parser that can be used to create other simple examples. The instructor should be aware that SimPar is not a realistic parallel language and is very limited in ability. Thus it should not be used for creating extended examples beyond CS1. During the walk-through of the example, students are asked to list the order in which the statements will execute on the processor. A parallel directive is then inserted for the two assignment statements and the meaning is explained to the students. The program is then extended to include array assignments instead of just simple assignments. This program is compiled and executed and the result examined in class. Students are then asked to comment on what other statements could be parallelized. The instructor leads them to an example where the result statement is put in the `PARALLEL` block along with the two assignment statements. This program is run, potentially several times, and the error demonstrated to the students. The students are then asked to describe the problem in the code. This is followed by a discussion of data dependence and the challenges with parallel programming.

Parallel Programming Tools

Students are told that SimPar is not a real language. The syntax for real languages are more complex and so is the programming model. Some of the currently available parallel languages and tools, including OpenMP, Pthreads, MPI are presented. The suitability of each is *briefly discussed*. The slides include example codes for each of these parallel languages. However, students are told they are not expected to learn the syntax at this stage.

Performance Metrics

In this segment of the module, performance issues in parallel computing are reiterated. This is followed by definitions and examples of sequential and parallel performance metrics. A simple parallel search code written in SimPar is used to do an in-class demo to show the differences in the performance metrics. Sequential and parallel (OpenMP) versions of the code are also shown in class. The code is compiled and executed with different data sets. Execution time and energy are measured for each run. A convenient tool for measuring power consumption on Intel processors is Likwid [4], freely available for download. The specific performance metrics and definitions that are discussed include

- Execution time
- Energy
- Speedup and Greenup
- Amdahl's Law
- Linear speedup
- Scalability

Pedagogical Notes

The author has used this module in CS1 courses in three semesters at Texas State University. In all three cases, it was helpful to introduce this module towards the end of the semester when students are somewhat more confident with the syntax of the sequential language that is being used in the class.

For the in-class activity, we found that a group size of four and a section size of two pages per member for the most *parallel* group is ideal. Making groups larger, makes the *sequential* group not as engaged. More than two pages of dense text makes the example run too long. We also found that, it is helpful to assign some form of reward to the team finishing first. This motivates the teams to be more engaged in the activity. Our experience also showed that it is better to place the stronger and more vocal students in the sequential group. Since the activity is framed as a competition and the sequential group is almost certain to not win, putting underperforming students in that group is not advisable.

It is advisable that instructors practice the live coding examples ahead of lecture time. Students often raise questions and suggest alternate approaches. The instructor should be fairly comfortable with the examples in order to incorporate these suggestions on-the-fly. The instructor should also take care to use the same system for the demo as the one used for practice. Variations in system configuration can make some examples not work as expected.

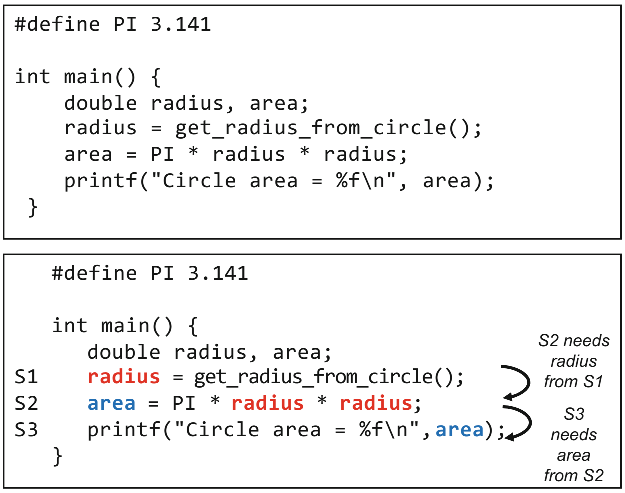
Sample Exercises

1. Computer A has 4 processors and Computer B has 8 processors. A parallel program P, takes 16 s to run on A and 12 s to run on B. Is this the type of performance you would expect out of P? Give one explanation as to why P does not achieve more/less performance.
2. Execute simple programs written in SimPar. Compare their performance with performance of sequential versions.
3. Download the C++ implementations of (i) knapsack and (ii) quicksort from `http://tues.cs.txstate.edu`. Consider the opportunities for parallelism in these two codes. Insert SimPar directives to parallelize the two applications. Execute the parallel applications and compare their performance with the sequential version of the code.

Task Orchestration

This module focuses on performance issues related to communication and synchronization of parallel applications. It is intended to be introduced in the Operating Systems course, as it provides the most context for the material covered.

Recommended Length 1.5 lectures (2 h)
Recommended Course Operating Systems



```c
#define PI 3.141

int main() {
    double radius, area;
    radius = get_radius_from_circle();
    area = PI * radius * radius;
    printf("Circle area = %f\n", area);
 }
```

```c
   #define PI 3.141

   int main() {
      double radius, area;
S1    radius = get_radius_from_circle();
S2    area = PI * radius * radius;
S3    printf("Circle area = %f\n",area);
   }
```

Fig. 4 Code example illustrating data dependence

Organization and Content

This module begins by introducing students to some fundamental concepts in parallel programming. Notions of data dependence, synchronization, race condition, load balance and task granularity are explained. Architecture-specific performance issues such as those that occur on shared and distributed-memory parallel computers are also covered. A producer-consumer application is used as a running example to illustrate various performance issues. Power-performance trade-offs are highlighted in each context.

Data Dependence

After a quick review of parallel computing (two slides, as used in CS1 module), the module introduces the students to the notion of data dependence in parallel programs. Sequential and parallel versions of a simple function is presented. The example in Fig. 4 uses the computation of an area of a circle. But many other examples are possible. The parallel version of the example code is written in SimPar [3].

Students are asked to predict the outcome of the code when executed with certain input. The code is run several times in sequential and parallel mode. Results are discussed and students are asked to comment on the discrepancy. Following this discussion, the annotated code is presented as a slide, highlighting the dependencies in the code. The formal definition of data dependence is then presented. Various forms of dependence are also discussed briefly. The point is

Fig. 5 Sequential version of producer-consumer code

```
int main() {
   while (!done) {
      fill_buffer(buf);          // produce
      if (buf_is_full(buf))
         empty_buffer(buf);      // consume
   }
}
```

```
#pragma omp parallel {
   #pragma omp section {
      fill_buffer(buf);
   }
   #pragma omp section {
      empty_buffer(buf);
   }
}
```

Fig. 6 Incorrectly parallelized producer-consumer code

made that both sequential and parallel programs must preserve all dependencies in the code for semantically correct execution. For sequential programs this is trivial since instructions are executed in program order. If students have already taken the Architecture course, then the notion of instruction-level parallelism (ILP) can be brought into this discussion. An example can be used to convey that the degree to which ILP can be performed is determined by the dependencies between the statements in question. Re-ordering transformations performed by compilers can also be discussed to further illustrate the importance of data dependence in semantically correct program execution.

Following this, the running example, a produce-consumer application is presented. The one shown in Fig. 5 uses the bounded-buffer problem as an example. But many other examples for parallel producer-consumer can be created with slight modifications. The supplementary material for this module includes an example with the knapsack problem. The sequential code is then explained to the class. (Figure 5 omits the actual producer-consumer functions). The parallel version of the code is then presented. Figure 6 shows the example in OpenMP. The instructor may continue the parallel example in SimPar but then it cannot be used later in the module for performance experiments, as the results would prove non-intuitive. If an OpenMP example is used, a brief review of OpenMP syntax may be required at this point. Alternatively, this can be handled off-line with the aid of tutorials or handouts, as discussed in section "Pedagogical Notes". The parallel version of the code is executed several times to produce incorrect results. Again, students are asked to identify the cause of the problem. Class discussion ensues, until the dependencies in the code have been identified and clearly articulated.

Fig. 7 Another incorrectly parallelized producer-consumer code

```
full = 0;
#pragma omp parallel {
   #pragma omp section {
      fill_buffer(buf);
      full = 1;
   }
   #pragma omp section {
      while (!full) {
          /* wait */
      }
      empty_buffer(buf);
   }
}
```

```
flag = 0;
#pragma omp parallel {
   #pragma omp section {
         fill_buffer(buf);
         #pragma omp flush
         flag = 1;
         #pragma omp flush(flag)
  }
   #pragma omp section {
        #pragma omp flush
        while (!flag)
        #pragma omp flush(flag)
            empty_buffer(buf);
   }
}
```

Fig. 8 Correctly parallelized producer-consumer code

Synchronization

After it has been established that the code in Fig. 6 is producing incorrect results due to data dependence violation, students are then asked if it is possible to correctly parallelize the code and if so what conditions must hold. This discussion leads to the notion of synchronization in parallel programs. The example in Fig. 7 is then constructed in-class by editing the example from Fig. 6. This code is compiled and executed several times to show the code still has not been correctly parallelized. The students are then asked to identify the dependence that caused this problem. This brings up the need for atomic operations, the idea of a critical section and race

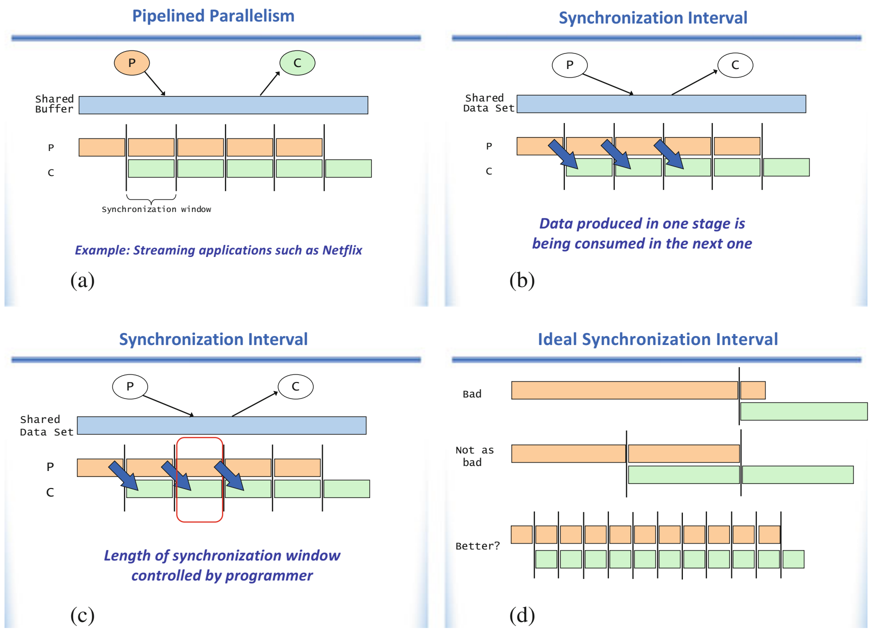

Pipelined Parallelism
P
C
Shared Buffer
P
C
Synchronization window
Example: Streaming applications such as Netflix
(a)
Synchronization Interval
P
C
Shared Data Set
P
C
Data produced in one stage is being consumed in the next one
(b)
Synchronization Interval
P
C
Shared Data Set
P
C
Length of synchronization window controlled by programmer
(c)
Ideal Synchronization Interval
Bad
Not as bad
Better?
(d)

Fig. 9 Lecture slides illustrating pipelined parallelism and the role of synchronization interval on performance

condition. The code is then fixed in-class by placing *guards* around the operations on the flag. This version of the code is shown in Fig. 8. Finally, the code is executed a few times to show that it indeed now produces correct results.

The pragmas are then modified to parallelize the example code in a pipelined fashion. Figure 9 shows a subset of the animated slides that explains pipelined-parallelism, the synchronization interval and its effect on performance.

Task Granularity

Task granularity and how it is controlled by the synchronization interval is introduced using a set of lecture slides. The impact of task granularity on performance is also explained. Following this the pipelined-parallel producer-consumer example is revisited. Students are asked to identify the amount of work performed per thread (i.e., task granularity). The amount of work is expressed in number of items read/written to the buffer. The code is then executed with different task granularity by using the BLOCK parameter in the OpenMP pragma. The results of these executions demonstrate to the student the significance of task granularity and cost of synchronization to parallel performance.

Load Balancing

OS scheduling is revisited to introduce the concept of load balancing. The basic scheduling algorithm is reviewed and once again the running example is used for an in-class demo. In this demo, the program is launched with multiple producers and consumers and the work is broken un-evenly between producers and consumers. At launch time, Linux `thread_affinity()` API is used to pin certain threads to specific cores to illustrate load imbalance. The script to perform this demo is available with the supplementary materials.

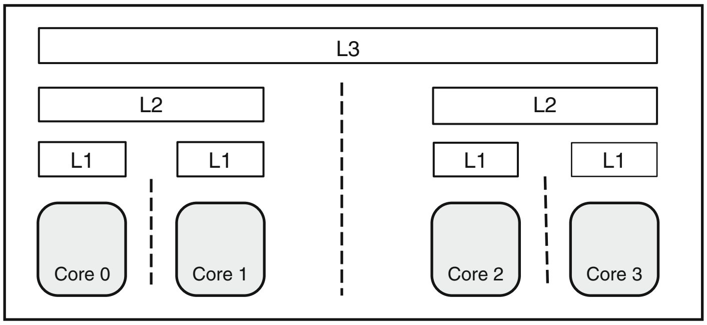
Pedagogical Notes

Although this module can be introduced in other upper-level courses (e.g., Unix Systems Programming), in our experience it works best in the OS course. A seamless integration is possible if the module is introduced in the OS class during the week when thread scheduling is discussed.

To provide background for OpenMP, a handout can be distributed ahead of time. A sample handout is included with the lecture material. Furthermore, there are several excellent online tutorials. Students can be asked to review one of these before the lecture. The supplementary material contains urls for online tutorials.

To increase student engagement, lecture slides related to load balancing for energy efficiency can be presented interactively as problem sets. The problems can be drawn out on the board or the slides can be animated and students can be asked to come up with a thread mapping solution as a group.

It is advisable that instructors practice the live coding examples ahead of lecture time. Students often raise questions and suggest alternate approaches. The instructor should be fairly comfortable with the examples in order to incorporate their suggestions into the demo.



Sample Exercises

1. Consider the high-level block diagram of a multicore system as shown in the figure above. A multi-threaded producer-consumer application is executing on this system. The application has 4 threads with 2 producers (p0 and p1) and 2 consumers (c2 and c1). Data produced by p0 is consumed by c1 and data produced by p1 is consumed by c0.
   - Describe a suitable schedule to improve the overall performance of the application. Explain why your schedule is likely to deliver improved performance.
   - Would your schedule change if the primary objective is to reduce power? Why or why not?
2. Implement a feedback queue scheduler using the OS framework used in the class. The scheduler should aim to minimize power consumption on a multicore system.
3. Parallelize the provided n-body simulation code using OpenMP and then derive an optimal affinity-based schedule. The scheduler can be implemented using affinity support in either Pthreads or GNU OpenMP.

Analysis and Evaluation

This module concentrates on performance estimation and measurement of parallel systems, including efficiency, linear and super-linear speedup, throughput, data locality, weak and strong scaling, and load balance. Performance estimation of sequential architectures and the implications of Amdahl's law are typically part of current computer architecture courses. This module extends these concepts and investigates parallel performance in light of Amdahl's law. It explores modern parallel benchmark suites such as PARSEC (task, data, and pipelined parallelism) and Lonestar (amorphous parallelism) and demonstrates how to write benchmark programs to measure the performance of parallel hardware. It discusses how to identify potential for speedup as well as upper speedup bounds and performance obstacles.

Recommended Length 1 lecture (1:15 min)
Recommended Course Compilers, Computer Architecture, Upper-level CS elective

Organization and Content

This module starts with a review of elementary performance concepts and OpenMP syntax. This is followed by discussion of several advanced performance concepts.

The lecture slides for this module are complemented with a series of micro-benchmarks written in OpenMP. Alternate implementations in Pthreads are also provided in the supplementary materials. Each benchmark highlights a particular performance issue. Each benchmark will also include several *student* versions. The student versions expose parameters in the code that students can alter in various ways to impact the performance of the code. The student versions of the code also includes omitted code blocks that the students are expected to fill in as an exercise. A set of scripts measure various performance metrics including execution time, cache misses and processor power consumption.

Review of Elementary Performance Concepts

This section is similar to the module section described in section "Performance Metrics". The main difference is that the examples used are more involved and written in OpenMP.

Review of OpenMP Syntax

This segment of the module provides a quick review of basic OpenMP syntax and semantics. It is assumed the students are familiar with parallel program execution but not necessarily with any programming language. Therefore, this introduction is very basic. Only the `parallel` regions and `parallel for` constructs are covered. The goal is to give students enough knowledge for them to modify existing code but not necessarily for them to be able to write efficient parallel programs on their own. If a student comes in with OpenMP programming experience, this module is still very useful as it will train her to tune the OpenMP pragmas to extract better performance from her code. As was done with the task orchestration module, the OpenMP tutorial can also be done offline to save some lecture time.

Strong and Weak Scaling

The notion of scalability of parallel programs is introduced in this segment. The distinction between strong scaling and weak scaling is discussed. The code shown in Fig. 10 is used as a running example. The code is explained and then executed with 1, 2, 4, and 8 threads on an 8 core machine. Other configurations are feasible based on computer availability. Before each run of the code, students are asked to guess the execution time. As the code is written the program will achieve strong scaling on up to 16 cores on current-generation processors. To observe scaling effects beyond 16 cores, the data set needs to be >4 GB. This introduces NUMA effects and page faults that prevent the application from achieving linear speedup.

```
pixel *src_images = (pixel *) malloc(sizeof(pixel) * PIXELS_PER_IMG * IMGS);
pixel *dst_images = (pixel *) malloc(sizeof(pixel) * PIXELS_PER_IMG * IMGS);

initialize(src_images);

DATA_ITEM_TYPE gs;
omp_set_num_threads(THREADS);                 // fix number of threads
start = omp_get_wtime();
int i;
#pragma omp  parallel for private(i)
for (i = 0; i < IMGS; i++) {            // process images in parallel
  int img_index = i * PIXELS_PER_IMG;
  for (int k = 0; k < ITERS; k++) {
    for (unsigned j = img_index; j < img_index + PIXELS_PER_IMG; j++)
      dst_images[j] = (0.3 * src_images[j].r + 0.59 *
                      src_images[j].g + 0.11 * src_images[j].b;
  }
}
```

Fig. 10 Example parallel code to demonstrate scaling

The code in Fig. 10 is then used to conduct a weak scaling experiment. The data set size is increased progressively until performance stops to scale. How much the data set needs to be increased depends on the particular platform where the code is being run. On some machines, runs for larger data sets can take up several minutes. So this needs to be weighed in when doing the demo. However, the code is designed in a way such that on most machines, memory bound behavior will show up for runs that take no more than 30 s. Similar to the strong scaling demo, before each run students are polled for the execution time. Following these demos the notions of strong scaling and weak scaling are formalized. A set of lectures slides and charts illustrating scaling trends are used for this purpose.

Linear and Super Linear Speedup

The code from Fig. 10 is re-used to explain the concepts of linear and super-linear speedups. The single-threaded version is labeled as the baseline and then speedup is calculated for 2, 4, and 8 thread versions. The obtained speedup is correlated with the number of threads/cores and shown to match the definition of linear speedup. The image processing example code is then transformed using tiling to improve data locality, as shown in Fig. 11. If time permits, this can be done live in class, as the technique is explained. Otherwise the example can be created ahead of time. The tiled version of the code is re-run with 2, 4, and 8 threads to demonstrate super-linear speedup. The working set size is orchestrated to exceed most L2 caches on current generation processors. A tiling size of 16–24 would keep the working set in cache. Some trial and error may be necessary prior to the demo to determine the exact size.

```
pixel *src_images = (pixel *) malloc(sizeof(pixel) * PIXELS_PER_IMG * IMGS);
pixel *dst_images = (pixel *) malloc(sizeof(pixel) * PIXELS_PER_IMG * IMGS);

initialize(src_images);

#define TILESIZE 64

DATA_ITEM_TYPE gs;
omp_set_num_threads(THREADS);              // fix number of threads
start = omp_get_wtime();
int i;
#pragma omp  parallel for private(i)
for (i = 0; i < IMGS; i++) {           // process images in parallel
  int img_index = i * PIXELS_PER_IMG;
  for (usigned j = img_index; j < img_index + PIXELS_PER_IMG; j = j + TILESIZE)
    for (int k = 0; k < ITERS; k++) {
      for (unsigned jj = j; jj < j + TILESIZE; jj++)
        for (unsigned j = img_index; j < img_index + PIXELS_PER_IMG; j++)
           dst_images[j] = (0.3 * src_images[j].r + 0.59 *
                          src_images[j].g + 0.11 * src_images[j].b;
    }
}
```

Fig. 11 Tiled version of image processing parallel code used to demonstrate data locality effects

Latency vs. Bandwidth

The concepts of memory bandwidth and latency and their effects on parallel performance is discussed next. Sequential versions of the code in Fig. 12 are first used to demonstrate the importance of locality in performance. The code on the left exploits spatial locality while the code on the right does not. The parallelization of the two codes is then explained and the parallel versions of the codes are executed. A second example with a tiled computation is also introduced briefly to illustrate the notion of temporal locality and its impact on performance. This demo establishes the fact that parallelism alone cannot overcome limitations with memory locality. The code in Fig. 10 is then run with a larger data set where the data set is large enough to exceed the available memory bandwidth per socket. After the execution of the program, the point is reiterated that scalable performance can be limited by memory factors.

SMP vs. NUMA

The discussion on latency and bandwidth leads to a discussion in parallel architectures and the main considerations for programming such systems. This discussion is left at a very high-level and uses slides to illustrate the differences between the architectures. Programming models and tools for the different systems is also discussed. GPUs and heterogeneous systems architectures with CPUs and GPUs are also touched on.

Fig. 12 Parallel code with and without spatial exploited spatial locality

```
int main() {
   int **a;
   omp_set_num_threads(12);
   a = (int **) malloc(sizeof(int *) *
                                  DIMSIZE);
   int i,j;
   for (i = 0; i < DIMSIZE; i++)
      a[i] = (int *) malloc(sizeof(int) *
                                  DIMSIZE);
#pragma omp parallel for private(i,j)
   for (i = 0; i < DIMSIZE; i++)
      for (j = 0; j < DIMSIZE; j++)
         a[i][j] = 17;
  return 0;
}
```

```
int main() {
   int **a;
   omp_set_num_threads(12);
   a = (int **) malloc(sizeof(int *) *
                                  DIMSIZE);
   int i,j;
   for (i = 0; i < DIMSIZE; i++)
      a[i] = (int *) malloc(sizeof(int) *
                                  DIMSIZE);
#pragma omp parallel for private(i,j)
    for (j = 0; j < DIMSIZE; j++)
       for (i = 0; i < DIMSIZE; i++)
           a[i][j] = 17;
   return 0;
}
```

Power vs. Performance

This module ends with a discussion on energy efficiency of parallel applications. The importance of saving power and attaining high-performance at specified power budgets is explained.

Pedagogical Notes

It is advisable to run the experiments a few times before the actual in-class demo. This will allow the codes to *adapt* to the execution environment and the instructor will be able to make any necessary changes. Details on how to tune the parameters of the code so that they exhibit the expected behavior are provided with sample codes and scripts.

In the default configuration, the slowest code in the examples runs for a few seconds. This is done to not take up too much class time. Nonetheless, if time permits, the longer versions of the codes should be used as the performance differences make more of an impression on the students. During these long runs the instructor may further elaborate on the topics.

Sample Exercises

```
#include<stdlib.h>
#include<stdio.h>
#include <omp.h>

#define DIMSIZE 80

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

   omp_set_num_threads(THREADS);
   int dim = atoi(argv[1]);
   a = (int **) malloc(sizeof(int *) * dim);
   int i,j,k;
   for (i = 0; i < dim; i++)
       a[i] = (int *) malloc(sizeof(int) * dim);

   int BLOCK = 220;
   int jj;
   for (j = 1; j < dim; j = j + BLOCK)
#pragma omp parallel for private(i,j)
     for (k = 0; k < 100; k++)
        for (jj = j; jj < (j + BLOCK); j++)
           for (i = 1; i < dim; i++)
               a[i][j] = 17;
     return 0;
}
```

1. Set the DIMSIZE, THREADS and BLOCK variables in the above code to different values (select values based on class discussion) and execute the code on a server X with 8 cores and server Y with 16 cores. Record performance statistics using `perf`. Prepare a report and explain the performance variations you observe on the two machines.
2. Download the PARSEC benchmark suite (http://parsec.cs.princeton.edu). Select one application from the group: *canneal*, *dedup* and *streamcluster* and another

application from the group: *swaptions*, *bodytrack*, *facesim*. Conduct a performance study of the two selected applications on a compute server with at least 16 cores. Use the `parsecmgmt` package to execute the applications with input data sets: small, medium, large and native, and with different thread counts: 2, 4, 8, 16, 32 and 64. Record performance statistics using `perf`.

What are the main performance trends you observe? What does that say about the characteristics of the two selected programs? Relate the performance trends to scalability concepts discussed in this module and prepare a report.

References

1. The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM)/IEEE Computer Society, "Curriculum Guidelines for Undergraduate Degree Programs in Computer Science," 2013.
2. S. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, and J. Wu, "2012 NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version I," http://www.cs.gsu.edu/~tcpp/curriculum/, accessed: 2018-02-11.
3. A. Qasem, "SimPar : A macro language for introducing parallel concepts to CS 1 students," https://github.com/apanqasem/simpar.git, accessed: 2018-02-11.
4. J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments," in *39th International Conference on Parallel Processing Workshops*, 2010.