# Multicore & GPU Programming : An Integrated Approach
# Instructor's Manual

Gerassimos Barlas

February 2, 2016

# Contents

# Chapter 1

# Introduction

## Exercises

1. Study one of the top 10 most powerful supercomputers in the world. Discover:

   - What kind of operating system does it run?
   - How many CPUs/GPUs is it made of?
   - What is its total memory capacity?
   - What kind of software tools can be used to program it?

   **Answer**

   Students should research the answer by visiting the Top 500 site and -if available- the site of one of the reported systems.

2. How many cores are inside the top GPU offerings from NVidia and AMD? What is the GFlop rating of these chips?

   **Answer** N/A.

3. The performance of the most powerful supercomputers in the world is usually reported as two numbers *Rpeak* and *Rmax*, both in TFlops (tera floating point operations per second) units. Why is this done? What are the factors reducing performance from *Rpeak* to *Rmax*? Would it be possible to ever achieve *Rpeak*?

   **Answer**

   This is done because the peak performance is unattainable. Sustained, measured performance on specific benchmarks, is a better indicator of the true machine potential.

   The reason these are different is communication overhead.

   *Rpeak* and *Rmax* could never be equal. Extremely compute-heavy applications, that have no inter-node communications, could asymptotically approach *Rpeak* if they were to run for a very long time. A very long execution time is required to diminish the influence of the start-up costs.

4. A sequential application with a 20% part that must be executed sequentially, is required to be accelerated five-fold. How many CPUs are required for this task?

   **Answer**

   This requires the application of Amdahl's law. The part that can be parallelized is $\alpha = 1 - 20\% = 80\%$. The speedup predicted by Amdahl's law is *speedup* $= \frac{1}{1-\alpha+\frac{\alpha}{N}}$.

   Achieving a three-fold speedup requires that:

$$\frac{1}{1-\alpha+\frac{\alpha}{N}} = 3 \Rightarrow \frac{1}{0.2+\frac{0.8}{N}} = 3 \Rightarrow \frac{0.8}{N} = \frac{1}{3} - 0.2 \Rightarrow N = \frac{0.8}{\frac{1}{3}-0.2} = 6 \tag{1.1}$$

   Achieving a 5-fold speedup requires that:

$$\frac{1}{0.2+\frac{0.8}{N}} = 5 \Rightarrow N = \frac{0.8}{\frac{1}{5}-0.2} = \frac{0.8}{0} = \infty \tag{1.2}$$

   So, it is impossible to achieve a 5-fold speedup, according to Amdahl's law.

5. A parallel application running on 5 identical machines, has a 10% sequential part. What is the speedup relative to a sequential execution on one of the machines? If we would like to double that speedup, how many CPU would be required?

   **Answer**

   This requires the application of Gustafson-Barsis' law as the information relates to a parallel application. The parallel part is $\alpha = 1 - 10\% = 90\%$. The speedup over a single machine is *speedup* $= 1-\alpha+N\cdot\alpha = .1+5\cdot 0.9 = 4.6$.

   Doubling the speedup would require $.1 + N \cdot 0.9 = 9.2 \Rightarrow N = \frac{9.1}{0.9} = 10.1$ machines. As $N$ has to be an integer, we have to round-up to the closest integer, i.e. $N = 11$.

6. An application with a 5% non-parallelizable part, is to be modified for parallel execution. Currently on the market there are two parallel machines available: machine X with 4 CPUs, each CPU capable of executing the application in 1hr on its own, and, machine Y with 16 CPUs, with each CPU capable of executing the application in 2hr on its own. Which is the machine you should buy, if the minimum execution time is required?

   **Answer**

   As the information provided relates to a sequential application, we have to apply Amdahl's law. The execution time for machine X is:

$$t_X = (1-\alpha)T + \frac{\alpha T}{N} = 0.05 * 1hr + \frac{0.95 \cdot 1hr}{4} = 0.2875hr \tag{1.3}$$

   The execution time for machine Y is:

$$t_Y = (1-\alpha)T + \frac{\alpha T}{N} = 0.05 * 2hr + \frac{0.95 \cdot 2hr}{16} = 0.21875hr \tag{1.4}$$

So we should buy machine Y.

7. Create a simple sorting application that uses the mergesort algorithm to sort a large collection (e.g. $10^7$) of 32-bit integers. The input data and output results should be stored in files, and the I/O operations should be considered a sequential part of the application. Mergesort is an algorithm that is considered appropriate for parallel execution, although it cannot be equally divided between an arbitrary number of processors, as Amdahl's and Gustafson-Barsis' laws require.

   Assuming that this equal division is possible, estimate $\alpha$, i.e. the part of the program that can be parallelized, by using a profiler like `gprof` or `valgrind` to measure the duration of mergesort's execution relative to the overall execution time. Use this number to estimate the predicted speedup for your program.

   Does $\alpha$ depend on the size of the input? If it does, how should you modify your predictions and their graphical illustration?

   **Answer** N/A

8. A parallel application running on 10 CPUs, spends 15% of its total time, in sequential execution. What kind of CPU (how much faster) would we need to run this application completely sequentially, while keeping the same total time?

   **Answer**

   This is an application of Gustafson-Barsis' law. If $T$ is the parallel execution time on the 10 CPUs, the sequential execution time on a single one would be $T_s = (1-\alpha)T + N \cdot \alpha \cdot T$. The fast CPU should match the parallel time, i.e. $T_f = T$ which means if should be $\frac{T_s}{T_f} = (1-\alpha) + N \cdot \alpha = 0.15 + 10 \cdot 0.85 = 8.65$ times faster.

# Chapter 2

# Multicore and Parallel Program Design

1. Perform a 2D agglomeration step for the image convolution problem of Section 2.2. What is the resulting number of communication operations?

   **Answer**

   Assuming that we target a grid of task groups forming $K$ rows x $M$ columns and that $K$ and $M$ divide the corresponding dimensions evenly, each group will hold $\frac{IMGX}{M}\frac{IMGY}{K}$ tasks.

   The number of communication operations are:

   - Four for "internal" groups. There are $(K-1)(M-1)$ internal groups. Each group sends $\frac{IMGX}{M}$ pixel values to its top and bottom neighbors, and $\frac{IMGY}{K}$ pixels to its left and right neighbors.
   - Two for corner groups. There are four corner groups.
   - Three for "boundary" groups. There are $2(K-2)+2(M-2)$ boundary groups.

   The total data volume communicated is :

$$totalComm = (2\frac{IMGX}{M} + 2\frac{IMGY}{K})(K-1)(M-1) + \quad (2.1)$$

$$(2\frac{IMGX}{M} + \frac{IMGY}{K})2(M-2) + \quad (2.2)$$

$$(\frac{IMGX}{M} + 2\frac{IMGY}{K})2(K-2) + \quad (2.3)$$

$$(\frac{IMGX}{M} + \frac{IMGY}{K})4 \quad (2.4)$$

2. Perform the comparison between the 1D and 2D decompositions of the heat diffusion example in Section 2.3.3, by assuming that (a) half-duplex communication links are available and (b) n-port communications are possible, i.e. all communications can take place at the same time over all the links.

   **Answer**

(a) If half-duplex communication links where used, then the time spend on communication would be doubled:

$$comp_{1D} + comm_{1D} = \frac{N^2}{P} \cdot t_{comp} + 4 \cdot (t_{start} + t_{comm} N)$$

$$comp_{2D} + comm_{2D} = \frac{N^2}{P} t_{comp} + 8 \left( t_{start} + t_{comm} \cdot \frac{N}{\sqrt{P}} \right) \tag{2.5}$$

which means that

$$comm_{1D} + comp_{1D} < comm_{2D} + comp_{2D} \Rightarrow$$
$$t_{start} > t_{comm} N \left( 1 - \frac{2}{\sqrt{P}} \right) \tag{2.6}$$

So there is no change in the condition that favors 1D over 2D.

(b) In the n-port case, the communication time per time step will be:

$$comm_{1D} = t_{start} + t_{comm} N$$

$$comm_{2D} = t_{start} + t_{comm} \cdot \frac{N}{\sqrt{P}}$$

so the comparison between the two decompositions would be based on

$$comm_{1D} + comp_{1D} < comm_{2D} + comp_{2D} \Rightarrow \sqrt{P} < 1$$

which is obviously false. So, 2D is always better than 1D.

3. How would communication costs affect the pipeline performance? Derive variations of Equations 2.16 to 2.18 that take into account a constant communication overhead between the pipeline stages.

**Answer**

Let's assume that communication between the stages costs a fixed amount of time $t_c$. Then we would have an additional overall time of $t_c(N + M)$, assuming that the last stage also sends data back to whoever is controlling the execution. Thus:

$$t_{total} = \sum_{j=0}^{l-1} t_j + N \cdot t_l + \sum_{j=l+1}^{M-1} t_j + t_c(N + M)$$

The processing rate of the pipeline is:

$$rate = \frac{N}{\sum_{j=0}^{l-1} t_j + N \cdot t_l + \sum_{j=l+1}^{M-1} t_j + t_c(N + M)}$$

The latency of the pipeline is:

$$latency = \sum_{j=0}^{M-2} t_j + t_c M$$

4. The total number of tasks calculated in Section 2.4.5 for the parallel quicksort of Listing 2.8, is based on the best-case assumption that the input is split in equal halves by every call to the `PartitionData` function. What would be the result if the worst-case (i.e. one part gets $N-1$ elements and the other part 0) were considered?

   **Answer**

   If we assume that the `PartitionData` function produces a zero sized part and a part with $N-1$ elements, then:

   $$T(N) = \begin{cases} 0 & \text{if } N \leq THRES \\ 1 + T(N-1) & \text{if } N > THRES \end{cases}$$

   as one of the calls to `QuickSort` would always have no input.

   Backward substitution can provide the answer:

   $$T(N) = 1 + T(N-1) = 1 + 1 + T(N-2) = k + T(N-k)$$

   $T(N-k)$ can be eliminated when $N-k = THRES \Rightarrow k = N - THRES$. Substituting this value of $k$ in the previous equation yields:

   $$T(N) = N - THRES + T(THRES) = N - THRES$$

   as $T(THRES) = 0$. This is obviously a poor result. What this formula does not convey is that parallelism is also eliminated as a result: there is effectively no overlap between the generated tasks.

5. Use a simple problem instance (e.g. a small array of integers) to trace the execution of the parallel quicksort of Listing 2.8. Create a Gantt graph for the tasks generated, assuming an infinite number of compute nodes is available for executing them. Can you calculate an upper bound for the speedup that can be achieved?

   **Answer**

   Given $N$ input elements, `PartitionData` executes between $N-1$ and $N+1$ key comparisons. For simplicity we will assume that $N$ comparisons are done and that the array is evenly split into parts equal in size to $\frac{N-1}{2}$ (taking out the pivot element).

   The first initial task that starts the sorting operation, will perform $N$ comparisons, before spawning a task for $\frac{N-1}{2}$ elements and continuing with the remaining $\frac{N-1}{2}$. The comparisons are done exclusively inside the `PartitionData` function, which means we have the following sequence of task executions and corresponding comparison steps:

   - 1 task, $N$ steps
   - 2 tasks, $\frac{N-1}{2}$ steps each
   - $2^2$ tasks, $\frac{N-2^2+1}{2^2}$ steps each
   - $2^3$ tasks, $\frac{N-2^3+1}{2^3}$ steps each
   - And so on...

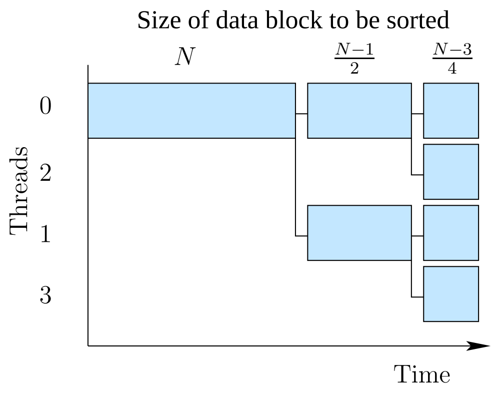


Figure 2.1: The first three steps (assuming an ideal scenario of a perfect partition each time) in the parallel quicksort algorithm of Listing 2.8.

Figure 2.1 illustrates this process.

So, the overall duration in number of comparisons is :

$$totalComp = \sum_{i=0}^{L-1} \frac{N - 2^i + 1}{2^i} + THRES$$

where $L$ is associated with when the size become smaller or equal to $THRES$ and no more tasks are spawned.

The value of $L$ is:

$$\frac{N - 2^L + 1}{2^L} = THRES \Rightarrow N - 2^L + 1 = 2^L THRES \Rightarrow$$
$$2^L = \frac{N+1}{THRES+1} \Rightarrow L = lg(\frac{N+1}{THRES+1})$$

Thus:

$$totalComp = \sum_{i=0}^{L-1} \frac{N - 2^i + 1}{2^i} + THRES =$$
$$(N+1)\sum_{i=0}^{L-1}(\frac{1}{2})^i - \sum_{i=0}^{L-1} 1 + THRES =$$
$$(N+1)\frac{(\frac{1}{2})^L - 1}{\frac{1}{2} - 1} - L + THRES =$$
$$2(N+1)(1 - \frac{THRES+1}{N+1}) - L + THRES =$$
$$2(N+1) - 2(THRES+1) - L + THRES =$$
$$2N - THRES - L \approx 2N$$

So the maximum speedup that could be ever achieved, given that the sequential quicksort performs $NlgN$ comparisons, is:

$$speedup_{max} = \frac{NlgN}{2N} = \frac{lgN}{2}$$

# Chapter 3

# Shared-memory programming : Threads

## Exercises

1. Enumerate and create the other timing diagrams that show the alternatives of Figure 3.4, when it comes to the final balance of the bank account.

   **Answer**

   All the possible permutations of the four events are allowed, as long as 1 proceeds 3 and 2 proceeds 4. So we have:

   - 1, 2, 3, 4 : produces wrong result
   - 1, 3, 2, 4 : produces correct result
   - 1, 2, 4, 3 : produces wrong result
   - 2, 1, 3, 4 : produces wrong result
   - 2, 1, 4, 3 : produces wrong result
   - 2, 4, 1, 3 : produces correct result

2. Research the term "fork bomb" and write a program that performs as such.

   **Answer**

```
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>

int main (int argc, char **argv)
{
  int N = atoi (argv[1]);

  for(int i=0;i<N;i++)
     pid_t cID = fork();

  sleep(INT_MAX);
  return 0;
}
```

3. Modify the producer-consumer example shown in Listing 3.11, so that the threads terminate after the number 100 is generated.

   **Answer**

   The only modification required affects the `consume` method, where the detection of the exit condition is done. It should become:

```
bool consume(int i) {
    // to be implemented
    cout << "@"; // just to show something is happening
    if (i == 100) return true;
    else return false;
}
```

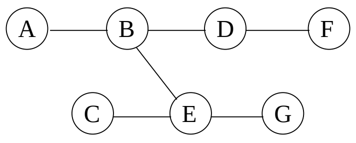
4. Suggest a modification to the program of Listing 3.12 so that the `IntegrCalc` threads can use any function that return a double and takes a double as a parameter.

   **Answer**

   A pointer to such a function needs to be passed to the `initClass` method, so that it can be called by the threads. The modifications are highlighted in the following code:

```
. . .
//---------------------------------------------------
// acts as a consumer
class IntegrCalc : public QThread {
private:
    int ID;
    static QSemaphore *slotsAvail;
    static QSemaphore *resAvail;
    static QMutex l2;
    static QMutex resLock;
    static Slice *buffer;
    static int out;
    static double *result;
    static double (*f)(double);    // <=======
    static QSemaphore numProducts;
public:
    static void initClass(QSemaphore *s, QSemaphore *a, Slice *b,↩
         double *r, double (*f)(double));

    IntegrCalc(int i) : ID(i) {
    };
    void run();
};
//---------------------------------------------
. . .
double (* IntegrCalc::f)(double);    // <=======

//---------------------------------------------

void IntegrCalc::initClass(QSemaphore *s, QSemaphore *a, Slice *b↩
    , double *res, double (*g)(double)) {  // <=======
    slotsAvail = s;
    resAvail = a;
    buffer = b;
    result = res;
    *result = 0;
    f = g ;  // <=======
}
//---------------------------------------------

void IntegrCalc::run() {
    while (1) {
        resAvail->acquire(); // wait for an available item
        l2.lock();
        int tmpOut = out;
        out = (out + 1) % BUFFSIZE; // update the out index
        l2.unlock();

        // take the item out
        double st = buffer[tmpOut].start;
        double en = buffer[tmpOut].end;
        double div = buffer[tmpOut].divisions;

        slotsAvail->release(); // signal for a new empty slot

        if (div == 0) break; // exit

        //calculate area
        double localRes = 0;
        double step = (en - st) / div;
        double x;
        x = st;
        localRes = f(st) + f(en);    // <=======
        localRes /= 2;
        for(int i=1; i< div; i++)   {
            x += step;
            localRes += f(x);    // <=======
          }
        localRes *= step;

        // add it to result
        resLock.lock();
        *result += localRes;
        resLock.unlock();
    }
}
//-----------------------------------------

int main(int argc, char *argv[]) {
    if (argc == 1) {
        cerr << "Usage " << argv[0] << " #threads #jobs\n";
        exit(1);
    }
    int N = atoi(argv[1]);
    int J = atoi(argv[2]);
    Slice *buffer = new Slice[BUFFSIZE];
    QSemaphore avail, buffSlots(BUFFSIZE);
    int in = 0;
    double result;

    IntegrCalc::initClass(&buffSlots, &avail, buffer, &result, ↩
        func);   // <=======

. . .
```

5. In a remote region of Siberia there are single tracks joining railroad stations. Obviously only one train can use a piece of track between two stations. The other trains can wait at the stations before they do their crossings. The following graph indicates the track and station layout:



Write a Qt program that simulates the journey of 3 trains with the following schedules:

- $A \to B \to E \to C$
- $D \to B \to E \to G$
- $C \to E \to B \to D \to F$

As each trains arrives at a station display a relative message. You can assume that a station can hold any number of trains waiting.

**Answer**

The `Train` class constructor expects two vectors : one containing the names of the stations that will be traversed, and one containing references to the binary semaphores controlling access to the lines connecting the stations.

```
#include <QThread>
#include <QMutex>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Train : public QThread
{
  private:
    int ID;
    vector<string> *stations;
    vector<QMutex *> *lineLocks;
  public:
    Train(int i, vector<string> *s, vector<QMutex *> *l) : ID(i),↩
         stations(s), lineLocks(l){}
    void run();
};

void Train::run()
{
    int i;
    for(i=0;i<lineLocks->size();i++)
    {
        cout << "Train " << ID << " is in station " << stations->↩
            at(i) << endl;
        lineLocks->at(i)->lock();
        cout << "Train " << ID << " is traveling to station " << ↩
            stations->at(i+1) << endl;
        lineLocks->at(i)->unlock();
    }
    cout << "Train " << ID << " arrived in station " << stations↩
        ->at(i) << endl;
}


int main(int argc, char *argv[])
{
    QMutex AB, BD, DF, BE, CE, EG; // mutices for the lines
    // one for each train
    vector<QMutex *> routes[3];
    vector<string> st[3];

    routes[0].push_back(&AB);
    routes[0].push_back(&BE);
    routes[0].push_back(&CE);
    st[0].push_back("A");
    st[0].push_back("B");
    st[0].push_back("E");
    st[0].push_back("C");

    routes[1].push_back(&BD);
    routes[1].push_back(&BE);
    routes[1].push_back(&EG);
    st[1].push_back("D");
    st[1].push_back("B");
    st[1].push_back("E");
    st[1].push_back("G");

    routes[2].push_back(&CE);
    routes[2].push_back(&BE);
    routes[2].push_back(&BD);
    routes[2].push_back(&DF);
    st[2].push_back("C");
    st[2].push_back("E");
    st[2].push_back("B");
    st[2].push_back("D");
    st[2].push_back("F");

    // thread spawning
    Train *t[3];
    for(int i=0;i<3;i++)
    {
        t[i] = new Train(i, &st[i], &routes[i]);
        t[i]->start();
    }

    for(int i=0;i<3;i++)
        t[i]->wait();
    return 0;
}
```

6. Modify the program of the previous exercise so that each station can hold only 2 trains. Can this lead to deadlocks?

   If you have not done so already, make sure that your program uses only one thread class.

   **Answer**

   This cannot lead to deadlocks as long as we have three or less trains. For four or more we could have pairs of trains trying to cross opposite directions of the same line, leading to deadlock.

   The only required changes involves controlling access to the stations via a set of general semaphores.

```
#include <QThread>
#include <QMutex>
#include <QSemaphore>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Train : public QThread
{
private:
    int ID;
    vector<string> *stations;
    vector<QSemaphore *> *stationsCtrl;
    vector<QMutex *> *lineLocks;
public:
    Train(int i, vector<string> *s, vector<QMutex *> *l, vector<↩
        QSemaphore *> *c) : ID(i), stations(s), lineLocks(l), ↩
        stationsCtrl(c){}
    void run();
};

void Train::run()
{
    int i;
    stationsCtrl->at(0)->acquire();
    for(i=0;i<lineLocks->size();i++)
    {
        cout << "Train " << ID << " is in station " << stations->↩
            at(i) << endl;
        stationsCtrl->at(i+1)->acquire(); // reserve station ↩
            before locking the line

        lineLocks->at(i)->lock();
        cout << "Train " << ID << " is traveling to station " << stations->at(i+1) << endl;
        lineLocks->at(i)->unlock();

        stationsCtrl->at(i)->release(); // release previous station reservation
        cout<<ID <<" "<< stationsCtrl->at(i)->available() << endl;
    }
    cout << "Train " << ID << " arrived in station " << stations->at(i) << endl;
}


int main(int argc, char *argv[])
{
    QMutex AB, BD, DF, BE, CE, EG;
    QSemaphore A(2), B(2), C(2), D(2), E(2), F(2), G(2);
    // one for each train
    vector<QMutex *> routes[3];
    vector<QSemaphore *> stCapac[3];
    vector<string> st[3];

    routes[0].push_back(&AB);
    routes[0].push_back(&BE);
    routes[0].push_back(&CE);
    st[0].push_back("A");
    st[0].push_back("B");
    st[0].push_back("E");
    st[0].push_back("C");
    stCapac[0].push_back(&A);
    stCapac[0].push_back(&B);
    stCapac[0].push_back(&E);
    stCapac[0].push_back(&C);

    routes[1].push_back(&BD);
    routes[1].push_back(&BE);
    routes[1].push_back(&EG);
    st[1].push_back("D");
    st[1].push_back("B");
    st[1].push_back("E");
    st[1].push_back("G");
    stCapac[1].push_back(&D);
    stCapac[1].push_back(&B);
    stCapac[1].push_back(&E);
    stCapac[1].push_back(&G);

    routes[2].push_back(&CE);
    routes[2].push_back(&BE);
    routes[2].push_back(&BD);
    routes[2].push_back(&DF);
    st[2].push_back("C");
    st[2].push_back("E");
    st[2].push_back("B");
    st[2].push_back("D");
    st[2].push_back("F");
    stCapac[2].push_back(&C);
    stCapac[2].push_back(&E);
    stCapac[2].push_back(&B);
    stCapac[2].push_back(&D);
    stCapac[2].push_back(&F);

    Train *t[3];
    for(int i=0;i<3;i++)
    {
        t[i] = new Train(i, &st[i], &routes[i], &stCapac[i]);
        t[i]->start();
    }

    for(int i=0;i<3;i++)
        t[i]->wait();
    return 0;
}
```

7. A desktop publishing (like PageMaker) application has two threads running: one for running the GUI and one for doing background work. Simulate this application in Qt. Your implementation should have the thread corresponding to the GUI, send requests to the other thread to run tasks on its behalf. The tasks should be (obviously just printing a message is enough for the simulation):

   - Printing
   - Mail merging
   - PDF generation

   After performing each requested task, the second thread should wait for a new request to be send to it. Make sure that the first thread does not have to wait for the second thread to finish before making new requests.

   **Answer**

   The following solution is a producer-consumer derivative, where just two counting semaphores are needed for each pair of interacting threads. There are two buffers, one for handling the interaction between GUI and mail-merge and one for the interaction between GUI and PDF generation.

```
#include <QSemaphore>
#include <QThread>
#include <stdlib.h>
#include <unistd.h>
#include <iostream>

using namespace std;

const int TERMINFLAG = -1;
const int RUNS = 20;
const int BUFFSIZE = 5;

// the following should not be global variables
// but they are so to reduce the length of the code
QSemaphore mailReq (0), mailSpace (BUFFSIZE);
int mailJob[BUFFSIZE];          // buffer
int m_in = 0, mout = 0;

QSemaphore pdfReq (0), pdfSpace (BUFFSIZE);
int pdfJob[BUFFSIZE];           // buffer
int pin = 0, pout = 0;
//================================================
class GUI:public QThread
{
public:
  void run ();
};
//---------------
void GUI::run ()
{
  for (int i = 0; i < RUNS; i++)
    {
      int choice = rand () % 2;
      if (choice)               // mail merge
        {
          mailSpace.acquire ();
          mailJob[m_in] = i;
          m_in = (m_in + 1) % BUFFSIZE;
          mailReq.release ();
        }
      else                      // PDF generation
        {
          pdfSpace.acquire ();
          pdfJob[pin] = i;
          pin = (pin + 1) % BUFFSIZE;
          pdfReq.release ();
        }
    }
  // term_ination
  mailSpace.acquire ();
  mailJob[m_in] = TERMINFLAG;
  mailReq.release ();

  pdfSpace.acquire ();
  pdfJob[pin] = TERMINFLAG;
  pdfReq.release ();
}

//===============================================
class MailMerge:public QThread
{
public:
  void run ();
};
//---------------
void MailMerge::run ()
{
  while (1)
    {
      mailReq.acquire ();
      int job = mailJob[mout];
      mout = (mout + 1) % BUFFSIZE;
      if (job == TERMINFLAG)
        break;
      cout << "Mail merge job #" << job << endl;
      mailSpace.release ();
    }
}

//===============================================
class PDFGen:public QThread
{
public:
  void run ();
};
//---------------
void PDFGen::run ()
{
  while (1)
    {
      pdfReq.acquire ();
      int job = pdfJob[pout];
      pout = (pout + 1) % BUFFSIZE;
      if (job == TERMINFLAG)
        break;
      cout << "PDF generation job #" << job << endl;
      pdfSpace.release ();
    }
}

//===============================================
int main (int argc, char *argv[])
{
  GUI g;
  PDFGen p;
  MailMerge m;
  g.start ();
  p.start ();
  m.run ();                     // m.start();  // can use run instead of start to avoid starting an extra thread

  g.wait ();
  p.wait ();
  // m.wait(); // see above
  return 0;
}
```

8. A popular bakery has a baker that cooks a loaf of bread at a time and

deposits it on a counter. Incoming customers pick up a loaf from the counter and exit the bakery. The counter can hold 20 loafs. If it is full the baker stops baking bread. If it is empty, a customer waits. Use semaphores to solve the coordination problem between the baker and the customers.

**Answer**

This is an instance of the producer-consumers problem. Because the baker thread is producing just simple integers in sequence, we can use a `QAtomicInt` variable for retrieving loaf IDs on the customers' side.

```
#include <QThread>
#include <QSemaphore>
#include <QAtomicInt>
#include <iostream>
#include <stdlib.h>

using namespace std;

const int SPACE=20;

QSemaphore loafs(0), counterSpace(SPACE);
QAtomicInt loafID;
//=================================
class Baker : public QThread
{
private:
    int totalBread;
public:
    Baker(int t) : totalBread(t){}
    void run();
};
//---------------------------------
void Baker::run()
{
   for(int i=0;i<totalBread;i++)
   {
       counterSpace.acquire();
       cout << "Baker baked # " << i << endl;
       loafs.release();
   }
}
//=================================
class Customer : public QThread
{
private:
    int ID;
public:
    Customer(int i) : ID(i){}
    void run();
};
//---------------------------------
void Customer::run()
{
   loafs.acquire();
   int myBread = loafID.fetchAndAddOrdered(1);
   counterSpace.release();
   cout << "Customer " << ID << " got bread #" << myBread << endl↩
       ;
}
//=================================
int main(int argc, char *argv[])
{
    int totalCustomers = atoi(argv[1]);
    Customer *c[totalCustomers];
    for(int i=0;i<totalCustomers;i++)
    {
        c[i]=new Customer(i);
        c[i]->start();
    }
    Baker b(totalCustomers);
    b.run();
61
62      // wait for termination
63      for(int i=0;i<totalCustomers;i++)
64          c[i]->wait();
65
66      return 0;
67  }
```

9. Because of customer demand, the bakery owner is considering the following enhancements to his shop:

   (a) Increase the capacity of the counter to 1000

   (b) Hire 3 more bakers

   Modify the solution of the previous exercise to accommodate these changes. Which is the easiest to implement?

   **Answer**

   (a) In this case, the only change required is to modify line 9 in the previous listing to:

```
const int SPACE=1000;
```

   (b) The complication in this case is the termination of the baker threads. For this purpose, the `totalCustomers` variable is used to initialize a general semaphore that is tested and decremented prior to the execution of each iteration of a baker thread (line 27):

```
1   #include <QThread>
2   #include <QSemaphore>
3   #include <QAtomicInt>
4   #include <iostream>
5   #include <stdlib.h>
6
7   using namespace std;
8
9   const int SPACE=20;
10
11  QSemaphore loafs(0), counterSpace(SPACE);
12  QSemaphore moreLoafs(0);
13  QAtomicInt loafID;
14  //==============================
15  class Baker : public QThread
16  {
17  private:
18      int ID;
19  public:
20      Baker(int i) : ID(i){}
21      void run();
22  };
23  //--------------------
24  void Baker::run()
25  {
26      int myProd=0;
27      while(moreLoafs.tryAcquire())
28      {
29          counterSpace.acquire();
30          loafs.release();
31          myProd++;
32      }
33      cout << "Baker #" << ID << " baked a total of " << myProd << ↩
            " breads" << endl;
34  }
35  //==============================
36  class Customer : public QThread
37  {
38  private:
    int ID;
public:
    Customer(int i) : ID(i){}
    void run();
};
//-------------------
void Customer::run()
{
    loafs.acquire();
    int myBread = loafID.fetchAndAddOrdered(1);
    counterSpace.release();
    cout << "Customer " << ID << " got bread #" << myBread << ↩
        endl;
}
//==============================
int main(int argc, char *argv[])
{
    int totalCustomers = atoi(argv[1]);
    Customer *c[totalCustomers];
    moreLoafs.release(totalCustomers);
    for(int i=0;i<totalCustomers;i++)
    {
        c[i]=new Customer(i);
        c[i]->start();
    }

    Baker *b[4];
    for(int i=0;i<4;i++)
    {
        b[i] = new Baker(i);
        b[i] -> start();
    }


    // wait for termination
    for(int i=0;i<totalCustomers;i++)
        c[i]->wait();
    for(int i=0;i<4;i++)
        b[i]->wait();

    return 0;
}
```

10. A bank account class is defined as follows:

```
class BankAccount {
  protected:
      double balance;
      string holderName;
  public:
      double getBalance();
      void deposit(double);
      void withdraw(double, int); // the highest the second ↩
          argument, the higher the priority of the request
};
```

Write the implementation of the three methods given above, so that withdraw operations are prioritized: if there are not enough funds in the account for all, the withdrawals must be done in order of priority regardless if there are some that can be performed with the available funds. You can assume that the priority level in the `withdraw` method is by default equal to 0, and that it can is upper bounded by a fixed constant `MAXPRIORITY`.

**Answer**

Solution is monitor based, using a separate wait condition for each level of priority.

```
// Monitor-based solution
#include <QThread>
```

```
#include <QMutexLocker>
#include <QWaitCondition>
#include <cstring>
#include <unistd.h>
#include <string>
#include <iostream>

using namespace std;

const int MAXPRIORITY = 10;
const int NUMAGENTS = 20;

//----------------------------------
class BankAccount
{
protected:
  double balance;
  string holderName;
  QMutex l;
  QWaitCondition priCond[MAXPRIORITY];
  int waitingCount[MAXPRIORITY];

public:
    BankAccount (string name, double init);
  double getBalance ();
  void deposit (double);
  void withdraw (double, int);  // the highest the second argument, the higher the priority of the request
};
//----------------------------------
BankAccount::BankAccount (string name, double init)
{
  holderName = name;
  balance = init;
  memset (waitingCount, 0, MAXPRIORITY * sizeof (int));
}

//----------------------------------
double BankAccount::getBalance ()
{
  return balance;
}

//----------------------------------
void BankAccount::deposit (double x)
{
  QMutexLocker ml (&l);
  balance += x;
  cerr << "Deposited " << x << endl;

  // now determine if any thread is waiting to withdraw
  int priLvl = MAXPRIORITY - 1;
  while (priLvl >= 0)
    {
      if (waitingCount[priLvl] > 0)
        {
          cerr << "Waking level " << priLvl << endl;
          priCond[priLvl].wakeAll ();  // wake up all the threads at that priority level
          return;
        }
      priLvl--;
    }
}

//----------------------------------
// the highest the second argument, the higher the priority of the request
void BankAccount::withdraw (double x, int lvl)
{
  QMutexLocker ml (&l);
  lvl = (lvl >= MAXPRIORITY) ? MAXPRIORITY - 1 : lvl;   //make sure priority is not too high

  // determine if others at the same or higher priority are waiting
  int othersWaiting = 0;
  int priLvl = MAXPRIORITY - 1;
  while (priLvl >= lvl)
    {
      othersWaiting += waitingCount[priLvl];
      priLvl--;
    }

  // if they are or the funds are not enough wait also
  if (othersWaiting > 0 || balance < x)
    {
      waitingCount[lvl]++;
      while (x > balance)
        priCond[lvl].wait (&l);

      balance -= x;
      waitingCount[lvl]--;
      if (waitingCount[lvl] == 0)
        {
          priLvl = lvl;
          while (priLvl >= 0 && waitingCount[priLvl] == 0)
            priLvl--;
          if (priLvl >= 0)
            priCond[priLvl].wakeAll ();
        }
    }
  else
    balance -= x;

  cerr << "Withdrew " << x << " with pri " << lvl << endl;
}

//--------------------------------------
// class for testing purposes
class Agent:public QThread
{
private:
  BankAccount * ba;
  double amount;
  int priority;
public:
  Agent (BankAccount * b, double x, int pri):ba (b), amount (x), priority (pri) {}
  void run ();
};

//--------------------------------------
void Agent::run ()
{
  if (amount > 0)
    ba->deposit (amount);
  else
    ba->withdraw (-amount, priority);
}

//--------------------------------------
int main (int argc, char *argv[])
{
  BankAccount b ("John Doe", 0);

  Agent *ag[NUMAGENTS];
  for (int i = 0; i < NUMAGENTS; i++)
    {
      ag[i] = new Agent (&b, -rand () % 20, i / 2);
      ag[i]->start ();
    }
  sleep (1);
  b.deposit (NUMAGENTS * 20); // make sure enough funds are available
  for (int i = 0; i < NUMAGENTS; i++)
    ag[i]->wait ();

  return 0;
}
```

11. The IT department of a big corporation is equipped with 5 high speed printers that are used by a multitude of threads. The threads are part of the same accounting process. Each of the threads is supposed to perform the following (pseudocode) sequence in order to printout any material:

```
...
printerID = get_available_printer();
// print to printerID printer
releasePrinter(printerID);
...
```

Write an appropriate implementation for the two functions listed above using semaphores. You can assume that the available printer IDs are stored in a shared buffer.

**Answer**

Solution is based on the producers-consumers pattern. A buffer with the available printer IDs is setup and managed. Function `get_available_printer()` operates as a consumer and function `releasePrinter()` operates as a producer. Only one counting semaphore is required as the buffer is as big as the overall number of printers.

```
#include <QThread>
#include <QMutex>
#include <QSemaphore>
#include <unistd.h>
#include <iostream>

using namespace std;

const int PRINTERS = 5;
const int NUMAGENTS = 20;

int printIDbuffer[PRINTERS]={0,1,2,3,4};
QSemaphore prnAvail(PRINTERS);
int in=0, out=0;
QMutex l1, l2;

//------------------------------------------
int get_available_printer()
{
   prnAvail.acquire();
   l1.lock();
   int tmp = printIDbuffer[out];
   out = (out +1)%PRINTERS;
   l1.unlock();
   return tmp;
}
//------------------------------------------
void releasePrinter(int printerID)
{
    l2.lock();
    printIDbuffer[in]=printerID;
    in = (in +1)%PRINTERS;
    l2.unlock();
    prnAvail.release();
}


//------------------------------------------
// class for testing purposes
class Agent:public QThread
{
private:
  int ID;
public:
  Agent (int i) : ID(i){}
  void run ();
};

//----------------------------------------
void Agent::run ()
{
  sleep(rand()%3);
  int printerID = get_available_printer();

  // print to printerID printer
  cout << ID << " is printing to " << printerID << endl;
  releasePrinter(printerID);
}

//----------------------------------------
int main (int argc, char *argv[])
{
  Agent *ag[NUMAGENTS];
  for (int i = 0; i < NUMAGENTS; i++)
    {
      ag[i] = new Agent (i);
      ag[i]->start ();
    }

  for (int i = 0; i < NUMAGENTS; i++)
    ag[i]->wait ();

  return 0;
}
```

12. Create 3 threads, each printing out the letters ’A’, ’B’ and ’C’. The printing must adhere to these rules:

    - The total number of ’B’s and ’C’s that have been output at any point in the output string cannot exceed the total number of ’A’s that have been output at that point.
    - After a ’C’ has been output, another ’C’ cannot be output until one or more ’B’s have been output.

    Use semaphores to solve the problem.

    **Answer**

```
#include <QThread>
#include <QMutex>
#include <QSemaphore>
#include <iostream>
#include <unistd.h>

using namespace std;

QSemaphore permit(0);
QMutex allowC;
volatile bool Bflag=true;

//----------------------------------------
class ThrA : public QThread
{
public:
    void run();
};
//------------------
void ThrA::run()
{
  while(1)
  {
      cout << 'A';
      permit.release();
  }
}

//---------------------------------------
class ThrB : public QThread
{
public:
    void run();
};
//------------------
void ThrB::run()
{
  while(1)
  {
      permit.acquire();
      cout << 'B';
      if(Bflag==false)
      {
          allowC.unlock();
          Bflag=true;
      }
  }
}
//---------------------------------------
class ThrC : public QThread
{
public:
    void run();
};
//------------------
void ThrC::run()
{
  while(1)
  {
      allowC.lock();
      permit.acquire();
      cout << 'C';
      Bflag=false;
  }
}
//---------------------------------------
int main (int argc, char *argv[])
{
  ThrA a;
  ThrB b;
  ThrC c;

  a.start();
  b.start();
  c.run();

  return 0;
}
```

13. Modify the previous exercise so that the printing is governed by this set of rules:

    - One 'C' must be output after two 'A's and three 'B's are output.
    - While there is no restriction on the order of printing 'A' and 'B', the corresponding threads must wait for a 'C' to be printed when the previous condition is met.

    Use a monitor to solve the problem.

    **Answer**

    The following solution uses the "critical section outside the monitor" approach.

```
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <QWaitCondition>
#include <iostream>
#include <unistd.h>

using namespace std;

class Monitor
{
private:
    int cntA, cntB;
    QMutex l;
    QWaitCondition cAB, cC;
public:
    Monitor() : cntA(0), cntB(0){}
    void allowA();
    void allowB();
    void allowC();
    void doneA();
    void doneB();
    void doneC();
};
//---------------
void Monitor::allowA()
{
   QMutexLocker ml(&l);
   while(cntA == 3)
       cAB.wait(&l);
}
//---------------
void Monitor::allowB()
{
    QMutexLocker ml(&l);
    while(cntB == 2)
        cAB.wait(&l);
}
//---------------
void Monitor::allowC()
{
    QMutexLocker ml(&l);
    while(cntA != 3 || cntB !=2)
        cC.wait(&l);
}
//---------------
void Monitor::doneA()
{
    QMutexLocker ml(&l);
    cntA++;
    if(cntA==3)
        cC.wakeOne();
}
//---------------
void Monitor::doneB()
{
    QMutexLocker ml(&l);
    cntB++;
    if(cntB==2)
        cC.wakeOne();
}
//---------------
void Monitor::doneC()
{
  QMutexLocker ml(&l);
  cntA = cntB = 0;
  cAB.wakeAll();
}
//--------------------------------
class ThrA : public QThread
{
private:
Monitor *m;
public:
    ThrA(Monitor *x) : m(x){}
    void run();
};
//----------------
void ThrA::run()
{
    while(1)
    {
        m->allowA();
        cout<<'A';
        m->doneA();
    }
}

//--------------------------------
class ThrB : public QThread
{
private:
Monitor *m;
public:
    ThrB(Monitor *x) : m(x){}
    void run();
};
//----------------
void ThrB::run()
{
      while(1)
      {
          m->allowB();
          cout<<'B';
          m->doneB();
      }
}
//--------------------------------
class ThrC : public QThread
{
private:
Monitor *m;
public:
    ThrC(Monitor *x) : m(x){}
    void run();
};
//----------------
void ThrC::run()
{
    while(1)
    {
        m->allowC();
        cout<<'C';
        m->doneC();
    }
}
//--------------------------------
int main (int argc, char *argv[])
{
  Monitor m;
  ThrA a(&m);
  ThrB b(&m);
  ThrC c(&m);

  a.start();
  b.start();
  c.run();

  return 0;
}
```

14. Address the termination problem in the previous exercise. How can the three threads terminate after e.g. a fixed number of As has been output? Or when a fixed total number of character has been output?

    **Answer**

The following code solves the termination problem for a fixed number of total characters. A character is reserved at the output before the termination of the `allow?()` methods, which compared to the previous exercise, return a boolean to indicate whether to continue execution or not. If the maximum number of characters has been reached, appropriate signals wake up any waiting threads.

```
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <QWaitCondition>
#include <iostream>
#include <unistd.h>

using namespace std;

class Monitor
{
private:
  int cntA, cntB, total;
  QMutex l;
  QWaitCondition cAB, cC;
public:
    Monitor (int t):cntA (0), cntB (0), total (t) {}
  bool allowA ();
  bool allowB ();
  bool allowC ();
  void doneA ();
  void doneB ();
  void doneC ();
};

//----------------
bool Monitor::allowA ()
{
  QMutexLocker ml (&l);
  while (cntA == 3 && total > 0)
    cAB.wait (&l);
  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeOne ();
      cC.wakeOne ();
      return false;
    }
}

//----------------
bool Monitor::allowB ()
{
  QMutexLocker ml (&l);
  while (cntB == 2 && total > 0)
    cAB.wait (&l);

  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeOne ();
      cC.wakeOne ();
      return false;
    }
}

//----------------
```

```
bool Monitor::allowC ()
{
  QMutexLocker ml (&l);
  while ((cntA != 3 || cntB != 2) && (total > 0))
    cC.wait (&l);

  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeOne ();
      cC.wakeOne ();
      return false;
    }
}

//-----------------
void Monitor::doneA ()
{
  QMutexLocker ml (&l);
  cntA++;
  if (cntA == 3)
    cC.wakeOne ();
}

//-----------------
void Monitor::doneB ()
{
  QMutexLocker ml (&l);
  cntB++;
  if (cntB == 2)
    cC.wakeOne ();
}

//-----------------
void Monitor::doneC ()
{
  QMutexLocker ml (&l);
  cntA = cntB = 0;
  cAB.wakeAll ();
}

//-----------------------------------
class ThrA:public QThread
{
private:
  Monitor * m;
public:
  ThrA (Monitor * x):m (x)
  {
  }
  void run ();
};

//-----------------
void ThrA::run ()
{
  while (m->allowA ())
    {
      cout << 'A';
      m->doneA ();
    }
}

//-----------------------------------
class ThrB:public QThread
{
private:
  Monitor * m;
public:
  ThrB (Monitor * x):m (x)
  {
  }
  void run ();
};

//----------------
void ThrB::run ()
{
  while (m->allowB ())
    {
      cout << 'B';
      m->doneB ();
    }
}

//----------------------------------
class ThrC:public QThread
{
private:
  Monitor * m;
public:
  ThrC (Monitor * x):m (x)
  {
  }
  void run ();
};

//----------------
void ThrC::run ()
{
  while (m->allowC ())
    {
      cout << 'C';
      m->doneC ();
    }
}

//----------------------------------
int main (int argc, char *argv[])
{
  Monitor m (100);
  ThrA a (&m);
  ThrB b (&m);
  ThrC c (&m);

  a.start ();
  b.start ();
  c.run ();

  a.wait ();
  b.wait ();
  return 0;
}
```

15. Create 4 threads, each printing out the letters 'A', 'B', 'C', and 'D'. The printing must adhere to these rules:

    - The total number of 'A's and 'B's that have been output at any point in the output string cannot exceed the total number of 'C's and 'D's that have been output at that point.
    - The total number of 'A's that have been output at any point in the output string cannot exceed twice the number of 'B's that have been output at that point.
    - After a 'C' has been output, another 'C' cannot be output until one or more 'D's have been output.

    Solve the problem using (a) semaphores and (b) a monitor.

**Answer**

(a)

```
 #include <QThread>
#include <QMutex>
#include <QSemaphore>
#include <iostream>
#include <unistd.h>

using namespace std;

QSemaphore allowAB (0), permitA (0), total (100);
QMutex permitC;
bool DisOut = true;

//------------------------------------
class ThrA:public QThread
{
public:
  void run ();
};

//----------------
void ThrA::run ()
{
  while (1)
    {
      permitA.acquire ();  // for rule #2
      allowAB.acquire ();  // for rule #1
      if (!total.tryAcquire ())
        break;
      cout << 'A';
      usleep (1);
    }
}

//------------------------------------
class ThrB:public QThread
{
public:
  void run ();
};

//----------------
void ThrB::run ()
{
  while (1)
    {
      allowAB.acquire (); // for rule #1
      if (!total.tryAcquire ())
        break;
      cout << 'B';
      permitA.release (2); // for rule #2
      usleep (1);
    }
}

//------------------------------------
class ThrC:public QThread
{
public:
  void run ();
};

//----------------
void ThrC::run ()
{
  while (1)
    {
      permitC.lock (); // for rule #3
      if (!total.tryAcquire ())
        break;
      cout << 'C';
      DisOut = false;
      allowAB.release (); // for rule #1
      usleep (1);
    }
}

//-----------------------------------
class ThrD:public QThread
{
public:
  void run ();
};

//----------------
void ThrD::run ()
{
  while (1)
    {
      cout << 'D';
      if (!total.tryAcquire ())
        break;
      allowAB.release (); // for rule #1
      if (DisOut == false) // for rule #3
        {
          DisOut = true;
          permitC.unlock ();
        }
      usleep (1);
    }

  // Make sure all terminate
  // This is the most appropriate thread to do this, as ThrD does not wait for any other
  allowAB.release (2);
  permitA.release ();
  if (DisOut == false)
    {
      permitC.unlock ();
    }

}

//-----------------------------------
int main (int argc, char *argv[])
{
  ThrA a;
  ThrB b;
  ThrC c;
  ThrD d;

  a.start ();
  b.start ();
  c.start ();
  d.run ();

  a.wait ();
  b.wait ();
  c.wait ();
  return 0;
}
```

(b) In this case, the `Monitor` class could be equipped with generic `allow()` and `done()` methods, that would accept as a parameter the character that is supposed to be printed, i.e. the type of thread calling. In the following code we stick with separate pairs of `Monitor` methods for each type of thread, to make the program easier to read.

```
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <QWaitCondition>
#include <iostream>
```

```
#include <unistd.h>

using namespace std;

class Monitor
{
private:
  int allowAB, permitA, total;
  bool permitC;
  QMutex l;
  QWaitCondition cAB, cA, cC;   // one condition for each of the ↩
      three rules
public:
    Monitor (int t):allowAB (0), permitA (0), total (t), permitC ↩
        (true)
  {
  }
  bool allowA ();
  bool allowB ();
  bool allowC ();
  bool allowD ();
  void doneA ();
  void doneB ();
  void doneC ();
  void doneD ();
};

//-----------------
bool Monitor::allowA ()
{
  QMutexLocker ml (&l);
  while (permitA == 0 && total > 0)
    cA.wait (&l);

  while (allowAB == 0 && total > 0)
    cAB.wait (&l);

  allowAB--;
  permitA--;
  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeOne ();
      cC.wakeOne ();
      return false;
    }
}

//-----------------
bool Monitor::allowB ()
{
  QMutexLocker ml (&l);
  while (allowAB == 0 && total > 0)
    cAB.wait (&l);

  allowAB--;
  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeOne ();
      cC.wakeOne ();
      return false;
    }
}

//-----------------
```

```
bool Monitor::allowC ()
{
  QMutexLocker ml (&l);
  while (permitC == false && total > 0)
    cC.wait (&l);

  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeAll ();
      cA.wakeOne ();
      return false;
    }
}

//--------------------
bool Monitor::allowD ()
{
  QMutexLocker ml (&l);
  if (total)
    {
      total--;
      return true;
    }
  else
    {
      cAB.wakeAll ();
      cA.wakeOne ();
      cC.wakeOne ();
      return false;
    }
}

//--------------------
void Monitor::doneA ()
{
  // nothing to be done here
}

//--------------------
void Monitor::doneB ()
{
  QMutexLocker ml (&l);
  permitA += 2;
  cA.wakeOne ();
}

//--------------------
void Monitor::doneC ()
{
  QMutexLocker ml (&l);
  allowAB++;
  permitC = false;
  cAB.wakeAll ();
}

//--------------------
void Monitor::doneD ()
{
  QMutexLocker ml (&l);
  if (permitC == false)
    {
      permitC = true;
      cC.wakeOne ();
    }
  allowAB++;
  cAB.wakeAll ();
}

//----------------------------------------
```

```
class ThrA:public QThread
{
private:
  Monitor * m;
public:
  ThrA (Monitor * x):m (x)
  {
  }
  void run ();
};

//-----------------
void ThrA::run ()
{
  while (m->allowA ())
    {
      cout << 'A';
      m->doneA ();
      usleep (1);
    }
}

//----------------------------------
class ThrB:public QThread
{
private:
  Monitor * m;
public:
  ThrB (Monitor * x):m (x)
  {
  }
  void run ();
};

//-----------------
void ThrB::run ()
{
  while (m->allowB ())
    {
      cout << 'B';
      m->doneB ();
      usleep (1);
    }
}

//----------------------------------
class ThrC:public QThread
{
private:
  Monitor * m;
public:
  ThrC (Monitor * x):m (x)
  {
  }
  void run ();
};

//-----------------
void ThrC::run ()
{
  while (m->allowC ())
    {
      cout << 'C';
      m->doneC ();
      usleep (1);
    }
}

//----------------------------------
class ThrD:public QThread
{
private:
  Monitor * m;
public:
  ThrD (Monitor * x):m (x)
  {
  }
  void run ();
};

//----------------
void ThrD::run ()
{
  while (m->allowD ())
    {
      cout << 'D';
      m->doneD ();
      usleep (1);
    }
}

//----------------------------------
int main (int argc, char *argv[])
{
  Monitor m (100);
  ThrA a (&m);
  ThrB b (&m);
  ThrC c (&m);
  ThrD d (&m);

  a.start ();
  b.start ();
  c.start ();
  d.run ();

  a.wait ();
  b.wait ();
  c.wait ();
  return 0;
}
```

16. Use semaphores to solve the typical cigarette smokers problem, where the agent signals directly the smoker missing the two ingredients placed on the table.

**Answer**

```
#include <QThread>
#include <QMutex>
#include <QSemaphore>
#include <iostream>
#include <stdlib.h>

using namespace std;

#define MAXSLEEP 1000
#define TOBACCO_PAPER 0
#define TOBACCO_MATHCES 1
#define PAPER_MATHCES 2

QSemaphore missingIngr[3];
QSemaphore  wakeAgent(0);
bool termFlag=false;
const char *msg[]={"having matches", "having paper", "having ↩
    tobacco"};
//*****************************************************
class Smoker : public QThread
{
  private:
    int missing;
  public:
    Smoker(int);
    void run();
};
//----------------------------------------------
Smoker::Smoker(int m) : missing(m){}
```

```
//-------------------------------------------------
void Smoker::run()
{
  while(1)
  {
     missingIngr[missing].acquire();  // wait for agent to send ↩
         signal
     if(termFlag) break;              // termination check
     cout << "Smoker " << msg[missing] << " is smoking\n";
     msleep(rand() % MAXSLEEP);
     wakeAgent.release();             // wake up agent
  }
}
//*************************************************
class Agent : public QThread
{
  private:
    int runs;
  public:
    Agent(int);
    void run();
};
//-------------------------------------------------
Agent::Agent(int r) : runs(r){}
//-------------------------------------------------
void Agent::run()
{
   for(int i=0;i<runs; i++)
   {
      int ingreds = rand() % 3;
      missingIngr[ingreds].release();
      wakeAgent.acquire();
   }
 // set termination flag and wake up all threads
 termFlag=true;
 missingIngr[0].release();
 missingIngr[1].release();
 missingIngr[2].release();
}
//*************************************************
int main(int argc, char **argv)
{
   Smoker *s[3];
   for(int i=0;i<3;i++)
   {
     s[i] = new Smoker(i);
     s[i]->start();
   }
   Agent a(atoi(argv[1]));
   a.run();

   for(int i=0;i<3;i++)
     s[i]->wait();

   return EXIT_SUCCESS;
}
```

17. Solve the cigarette smokers problem as described in Section 3.6.2, using semaphores.

**Answer**

In the following solution all smoker threads wake up and check the table contents.

```
#include <QThread>
#include <QMutex>
#include <QSemaphore>
#include <iostream>
#include <stdlib.h>

using namespace std;
```

```

#define MAXSLEEP 10
#define TOBACCO_PAPER 0
#define TOBACCO_MATHCES 1
#define PAPER_MATHCES 2

QSemaphore wakeUpSmoker[3];
QSemaphore wakeAgent(0);
volatile bool termFlag=false;
int table;
const char *msg[]={"having matches", "having paper", "having ↩
    tobacco"};
//***************************************************
class Smoker : public QThread
{
  private:
    int missing;
  public:
    Smoker(int);
    void run();
};
//---------------------------------------------
Smoker::Smoker(int m) : missing(m){}
//---------------------------------------------
void Smoker::run()
{
  while(1)
  {
     do
      {
       wakeUpSmoker[missing].acquire();
      }
     while(table != missing && !termFlag);

     if(termFlag) break;              // termination check
     cout << "Smoker " << msg[missing] << " is smoking\n";
     msleep(rand() % MAXSLEEP);
     wakeAgent.release();             // wake up agent
  }
}
//***************************************************
class Agent : public QThread
{
  private:
    int runs;
  public:
    Agent(int);
    void run();
};
//---------------------------------------------
Agent::Agent(int r) : runs(r){}
//---------------------------------------------
void Agent::run()
{
   for(int i=0;i<runs; i++)
   {
      table = rand() % 3;
      wakeUpSmoker[0].release();
      wakeUpSmoker[1].release();
      wakeUpSmoker[2].release();
      wakeAgent.acquire();
   }
 // set termination flag and wake up all threads
 termFlag=true;
 wakeUpSmoker[0].release();
 wakeUpSmoker[1].release();
 wakeUpSmoker[2].release();
}
//***************************************************
int main(int argc, char **argv)
{
   Smoker *s[3];
   for(int i=0;i<3;i++)
   {
      s[i] = new Smoker(i);
      s[i]->start();
    }
    Agent a(atoi(argv[1]));
    a.run();

    for(int i=0;i<3;i++)
      s[i]->wait();

    return EXIT_SUCCESS;
}
```

18. Model the movie-going process at a multiplex cinema using a monitor. Assume the following conditions:

    - There are 3 different movies played at the same time in 3 halls. The capacity of each hall is 4, 5 and 7 respectively.
    - One hundred customers are waiting to see a randomly chosen movie.
    - A cashier issues the tickets.
    - If a hall is full a movie begins to play.
    - A customer cannot enter a hall while a movie is playing or while the previous viewers are exiting the hall.
    - A movie will play for the last customers even if the corresponding hall is not full.

    **Answer**

```
#include <QThread>
#include <QMutex>
#include <QWaitCondition>
#include <iostream>
#include <vector>
#include <stdlib.h>

using namespace std;

const int NUMCUST=100;
//****************************************
class Monitor
{
private:
    int numHalls;
    vector<int> &capacity;
    int *inside;
    int *left;
    bool *finished;
    int numCustomersRemain;
    QMutex l;
    QWaitCondition *condE; // 'enter' array of conditions. One ↩
        for each hall
    QWaitCondition *condL; // 'leave' array of conditions
public:
    Monitor(vector<int> &cap, int numCust);
    ~Monitor();
    void enterTheater(int movie);
    void leaveTheater(int movie);
};
//----------------------------------------
Monitor::Monitor(vector<int> &cap, int numCust) : capacity(cap)
{
    numHalls=cap.size();
    numCustomersRemain = numCust;
    inside = new int[numHalls];
    for(int i=0;i<numHalls;i++) inside[i]=0;
    left = new int[numHalls];
    for(int i=0;i<numHalls;i++) left[i]=0;
    finished =  new bool[numHalls];
    for(int i=0;i<numHalls;i++) finished[i]=false;
    condE = new QWaitCondition[numHalls];
    condL = new QWaitCondition[numHalls];
}
//-----------------------------------
Monitor::~Monitor()
{
    delete[] inside;
    delete[] left;
    delete[] finished;
    delete[] condE;
    delete[] condL;
}
//-----------------------------------
void Monitor::enterTheater(int m)
{
    QMutexLocker ml(&l);
    while(inside[m] == capacity[m])
        condE[m].wait(&l);
    inside[m]++;
    if(inside[m] == capacity[m])   // is hall full?
    {
        cout << "Movie " << m << " started\n";
        finished[m]=true;          // set end-of-movie flag
        left[m]=0;
        condL[m].wakeAll();        // let all customers leave
    }

    numCustomersRemain--;
    if(numCustomersRemain==0) // start the remaining shows
    {
        for(int i=0;i<numHalls;i++)
            if(inside[i]>0)
            {
                cout << "Movie " << i << " started\n";
                finished[i]=true;
                left[i]=0;
                condL[i].wakeAll();
            }
    }
}
//-----------------------------------
void Monitor::leaveTheater(int m)
{
    QMutexLocker ml(&l);
    while(finished[m]==false)
        condL[m].wait(&l);
    left[m]++;
    if(left[m] == capacity[m])  // is hall empty
    {
        finished[m]=false;
        inside[m]=0;
        left[m]=0;
        condE[m].wakeAll();
    }
}
//****************************************
class Customer : public QThread
{
private:
    int ID;
    int movieID;
    Monitor *m;
public:
    Customer(int i, Monitor *c, int mID) : ID(i), movieID(mID), m↩
        (c) {};
    void run();
};
//-----------------------------------
void Customer::run()
{
    cout << "Customer " << ID << " wants to see " << movieID << "↩
```

```
        \n";
    m->enterTheater(movieID);
    cout << "Customer " << ID << " entered the theater\n";
    m->leaveTheater(movieID);
    cout << "Customer " << ID << " left the movies\n";
}
//*****************************************
int main(int argc, char *argv[])
{
    vector<int> c;
    c.push_back(4);
    c.push_back(5);
    c.push_back(7);

    Monitor m(c, NUMCUST);
    Customer *cust[NUMCUST];
    for(int i=0;i<NUMCUST;i++)
    {
        cust[i] = new Customer(i, &m, rand()%3);
        cust[i]->start();
    }

    for(int i=0;i<NUMCUST;i++)
        cust[i]->wait();

    return 0;
}
```

19. Write a multi-threaded password cracker based on the Producer-Consumer paradigm. The producer should generate plain-text passwords according to a set of rules and the consumers should be hashing each password and checking whether it matches a target signature. All the threads should terminate upon the discovery of a matching password. You can use the MD5 cryptographic hash function for this exercise.

**Answer**

The following listing shows a monitor based solution to the problem. There are two classes whose instances act as a producer (class `PassGenerator`) and consumers (class `PassChecker`).

The main program is expecting two command-line parameters : the number of password checkers and the maximum length of the brute-force generated passwords to be examined. Once a password with the same MD5 hash value is found, the **`finishUp`** method is called, which in turn terminates all remaining threads upon their next interaction with the Monitor.

```
#include <iostream>
#include <string>
#include <string.h>
#include "md5.h"
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <QWaitCondition>

using namespace std;

char *alphabet="1234567890abcdefghijklmnopqrstuvwxyz";
string target("5912d7bfd10f631f1715bf85bbb72d97");  // ciphertext
const int PASSSTORESIZE=100;  // size of password buffer
//************************************************
class Monitor
{
private:
  QMutex l;
  QWaitCondition full;
  QWaitCondition empty;
  char *buff[PASSSTORESIZE];
  int in, out, N, Ncons;
  int total;
  bool found;
public:
  Monitor(int consumers, int maxLen);
  ~Monitor();
  bool putCandidate(char *);  // both return false when it is ↩
      time to stop
  bool getCandidate(char *);
  void finishUp();
};
//----------------------------------
Monitor::Monitor(int consumers, int maxLen)
{
  Ncons = consumers;
  buff[0]= new char[(maxLen+1)*PASSSTORESIZE];
  for(int i=1;i<PASSSTORESIZE;i++)
    buff[i]= buff[0]+(maxLen+1)*i;
  in=0;
  out=0;
  N=0;
  total=0;
};
//----------------------------------
Monitor::~Monitor()
{
  delete [] buff[0];
}
//----------------------------------
void Monitor::finishUp()
{
  QMutexLocker ml(&l);
  found=true;
  empty.wakeAll();
  full.wakeOne();
}
//----------------------------------
bool Monitor::putCandidate(char *s)
{
  QMutexLocker ml(&l);
  while(N==PASSSTORESIZE && !found)
    full.wait(&l);
  strcpy(buff[in], s);
  in=(in+1)%PASSSTORESIZE;
  N++;
  total++;
  if(total%1000000==0)
    cout << "Checking #" << total << " : " <<  s << endl;
  empty.wakeOne();
  return !found;
}
//----------------------------------
bool Monitor::getCandidate(char *s)
{
  QMutexLocker ml(&l);
  while(N==0  && !found)
    empty.wait(&l);
  if(found) return false;

  strcpy(s, buff[out]);
  out=(out+1)%PASSSTORESIZE;
  N--;
  full.wakeOne();
  return !found;
}
//************************************************
class PassGenerator : public QThread
{
private:
  char *candidate;
  void recGen(int pos, int remain);
  Monitor *m;
  int maxLen;
  bool stopFlag;

public:
  PassGenerator(Monitor *x, int l);
  void run();
};
//-----------------------------------
PassGenerator::PassGenerator(Monitor *x, int l) : m(x), maxLen(l)↩
    , stopFlag(false)
{
  candidate = new char[maxLen+1];
}
//-----------------------------------
void PassGenerator::recGen(int pos, int remain)
 {
   if(remain==0)
   {
     candidate[pos]=0;
     stopFlag = ! m->putCandidate(candidate);
   }
   else
   {
     for(unsigned int i=0;i<strlen(alphabet) && !stopFlag;i++)
     {
       candidate[pos] = alphabet[i];
       recGen(pos+1, remain-1);
     }
   }
 }
//-----------------------------------
void PassGenerator::run()
{
   int i=6;
   while(stopFlag==false && i<=maxLen)
     recGen(0, i++);
}
//**************************************************
class PassChecker : public QThread
{
private:
  Monitor *m;
  int maxLen;
public:
  PassChecker(Monitor *x, int L) : m(x), maxLen(L){};
  void run();
};
//-----------------------------------
void PassChecker::run()
{
  char tmp[maxLen+1];
  while( m->getCandidate(tmp))
   {
    if(target == md5(tmp))
    {
      m->finishUp();
      cout << "Password : " << tmp << endl;
      break;
    }
  }
}
//**************************************************
 int main(int argc, char *argv[])
 {
   int Nchk = atoi(argv[1]);
   int L =    atoi(argv[2]);
   Monitor m(Nchk, L);
   PassChecker *chk[Nchk];
   PassGenerator gen(&m, L);

   gen.start();
   for(int i=0;i<Nchk;i++)
   {
     chk[i] = new PassChecker(&m,L);
     chk[i]->start();
  }

  gen.wait();
  for(int i=0;i<Nchk;i++)
    chk[i]->wait();

  return 0;
}
```

20. Write a multi-threaded program for finding the prime numbers in a user-supplied range of numbers. Compare the following design approaches:

    (a) Split the range in equal pieces and assign each one to a thread.

    (b) Have a shared `QAtomicInt` variable that holds the next number to be checked. Threads should read and increment this number before testing it.

    (c) Have a shared "monitor" object that returns upon request, a range of numbers to be tested. This can be considered a generalization of the previous design.

    Which of the designs is more efficient? Explain your findings.

    **Answer**

    (a) In the following listing, the `Repository` class is used to provide thread-safe storage for the discovered prime numbers. The `PrimeChecker` class is used to create threads that will check all the odd numbers in the range that is specified in their constructor. The main thread actually runs one of the `PrimeChecker` objects to avoid having to idle while they are doing useful work.

```
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <vector>

using namespace std;

//==================================================
// used for storing the results in a thread-safe manner
class Repository
{
private:
    vector<int> result;
    QMutex l;
public:
    void store(int i);
    vector<int> *getResult();
};
//--------------------------------------------------
void Repository::store(int i)
{
    QMutexLocker ml(&l);
    result.push_back(i);
}
//--------------------------------------------------
vector<int> *Repository::getResult()
{
  return &result;
}
//==================================================
class PrimeChecker :public QThread
```

```cpp
{
private:
    int first, last;
    Repository *repo;

    public:
    PrimeChecker(int f, int l, Repository *r) : first(f), last(l), repo(r){}
    void run();
};
//------------------------------------
void PrimeChecker::run()
{
    for(int i=first; i<=last; i+=2)
    {
        bool isPrime=true;
        int j=3;
        int limit = sqrt(i);
        while(j<= limit && isPrime)
        {
            if(i % j == 0)
                isPrime=false;
            j+=2;
        }
        if(isPrime)
            repo->store(i);
    }
}

//------------------------------------
int main(int argc, char *argv[])
{
    int numThreads;
    int a, b;
    a=atoi(argv[1]);
    b=atoi(argv[2]);
    numThreads=atoi(argv[3]);

    // make sure a and b are odd
    a = (a % 2 == 0) ? a+1 : a;
    b = (b % 2 == 0) ? b-1 : b;

    int rangePerThread = (b - a) * 1.0 / numThreads;
    // make sure rangePerThread is even
    rangePerThread = (rangePerThread % 2 == 0) ? rangePerThread : rangePerThread + 1;

    Repository rep;
    PrimeChecker *pc[numThreads];
    int first, last=a-2;
    for(int i=0;i<numThreads-1;i++)
    {
        first=last+2;
        last = first + rangePerThread;
        pc[i]= new PrimeChecker(first, last, &rep);
        pc[i]->start();
    }
    pc[numThreads-1] = new PrimeChecker(last+2, b, &rep);
    pc[numThreads-1]->run();

    for(int i=0;i<numThreads-1;i++)
        pc[i]->wait();

    cout << "Total primes found " << rep.getResult()->size() << endl;
    return 0;
}
```

To run the above program in order to find all the prime numbers in the range $[10^3, 10^8]$ using eight threads, one has to use the following command-line (the same conventions are used by the next versions as well):

```
$ primeChecker_V3 1000 100000000 8
```

(b) In this version, a shared `QAtomicInt` object is used to fetch odd number to check from primality. The `Repository` class serves the same purpose as above.

```
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <QAtomicInt>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <vector>

using namespace std;

//=============================================
// used for storing the results in a thread-safe manner
class Repository
{
private:
    vector<int> result;
    QMutex l;
public:
    void store(int i);
    vector<int> *getResult();
};
//---------------------------------------------
void Repository::store(int i)
{
    QMutexLocker ml(&l);
    result.push_back(i);
}
//---------------------------------------------
vector<int> *Repository::getResult()
{
  return &result;
}
//=============================================
class PrimeChecker :public QThread
{
private:
    int last;
    Repository *repo;
    QAtomicInt *next;

    public:
    PrimeChecker(QAtomicInt *n, int l, Repository *r) : last↩
        (l), repo(r), next(n){}
    void run();
};
//---------------------------------------------
void PrimeChecker::run()
{
    int candidate;
    while((candidate = next->fetchAndAddOrdered(2)) <= last)
    {
        bool isPrime=true;
        int j=3;
        int limit = sqrt(candidate);
        while(j<= limit && isPrime)
        {
            if(candidate % j == 0)
                isPrime=false;
            j+=2;
        }
        if(isPrime)
            repo->store(candidate);
    }
}

//---------------------------------------------
```

```
int main(int argc, char *argv[])
{
    int numThreads;
    int a, b;
    a=atoi(argv[1]);
    b=atoi(argv[2]);
    numThreads=atoi(argv[3]);

    // make sure a and b are odd
    a = (a % 2 == 0) ? a+1 : a;
    b = (b % 2 == 0) ? b-1 : b;

    QAtomicInt first(a);

    Repository rep;
    PrimeChecker *pc[numThreads];
    for(int i=0;i<numThreads-1;i++)
    {
        pc[i]= new PrimeChecker(&first, b, & rep);
        pc[i]->start();
    }
    pc[numThreads-1] = new PrimeChecker(&first, b, & rep);
    pc[numThreads-1]->run();

    for(int i=0;i<numThreads-1;i++)
        pc[i]->wait();

    cout << "Total primes found " << rep.getResult()->size()↩
         << endl;
    return 0;
}
```

The overall load balancing achieved by this version is substantially better: As the numbers get larger, the checking procedure becomes more time consuming. This makes the thread assigned the last portion of the range in the previous version, dominate the execution time.

(c) This version shares the load balancing efficiency of the second version, while reducing the cost of performing atomic operations. The **Monitor** effectively returns sets of 50 numbers to check, instead of the single one of the second version.

```
#include <QThread>
#include <QMutex>
#include <QMutexLocker>
#include <QAtomicInt>
#include <math.h>
#include <stdlib.h>
#include <iostream>
#include <vector>

using namespace std;

const int PARTLEN=100;
//=====================================
// used for storing the results in a thread-safe manner
class Repository
{
private:
    vector<int> result;
    QMutex l;
public:
    void store(int i);
    vector<int> *getResult();
};
//-------------------------------------
void Repository::store(int i)
{
    QMutexLocker ml(&l);
    result.push_back(i);
}
//---------------------------------------------
vector<int> *Repository::getResult()
{
    return &result;
}
//=============================================
class Monitor
{
private:
    int first, last;
    int currentAssign;
    int assignRange;
    QMutex l;
public:
    Monitor(int a, int b, int r) : first(a), last(b), ↩
        assignRange(r){currentAssign = a;}
    void getNextRange(int *a, int *b);
};
//---------------------------------------------
void Monitor::getNextRange(int *a, int *b)
{
    QMutexLocker ml(&l);
    if(currentAssign >= last) // terminating check
    {
        *a = -1;
        return;
    }
    *a = currentAssign;
    *b = currentAssign + assignRange;
    if(*b > last) *b = last;
    currentAssign = *b + 2;
}

//=============================================
class PrimeChecker :public QThread
{
private:
    Repository *repo;
    Monitor *mon;

public:
    PrimeChecker(Monitor *m, Repository *r) : repo(r), mon(m↩
        ){}
    void run();
};
//---------------------------------------------
void PrimeChecker::run()
{
    int first, last;
    while(1)
    {
        mon->getNextRange(&first, &last);
        if(first == -1) break;  // if the Monitor returns -1↩
             for first, exit
        for(int i=first; i<=last; i+=2)
        {
            bool isPrime=true;
            int j=3;
            int limit = sqrt(i);
            while(j<= limit && isPrime)
            {
                if(i % j == 0)
                    isPrime=false;
                j+=2;
            }
            if(isPrime)
                repo->store(i);
        }
    }
}

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

Table 3.1: Average execution times (in sec) over 10 runs, of the three prime checker variations. The best times in each case, are highlighted.

| | Threads | | | |
|---|---|---|---|---|
| **Version** | **1** | **2** | **4** | **8** |
| Even assignment to threads | **1.826** | 1.14 | 0.61 | 0.495 |
| `QAtomicInt` based | 2.048 | 1.04 | 0.54 | 0.426 |
| Monitor based | 1.837 | **0.92** | **0.46** | **0.418** |

```
{
    int numThreads;
    int a, b;
    a=atoi(argv[1]);
    b=atoi(argv[2]);
    numThreads=atoi(argv[3]);

    // make sure a and b are odd
    a = (a % 2 == 0) ? a+1 : a;
    b = (b % 2 == 0) ? b-1 : b;

    Monitor mon(a, b, PARTLEN);  // range of PARTLEN numbers↩
        , or PARTLEN/2 odd ones
    Repository rep;
    PrimeChecker *pc[numThreads];
    for(int i=0;i<numThreads-1;i++)
    {
        pc[i]= new PrimeChecker(&mon, & rep);
        pc[i]->start();
    }
    pc[numThreads-1] = new PrimeChecker(&mon, & rep);
    pc[numThreads-1]->run();

    for(int i=0;i<numThreads-1;i++)
        pc[i]->wait();

    cout << "Total primes found " << rep.getResult()->size()↩
         << endl;
    return 0;
}
```

The exact performance figures depend on the execution platform. On a Intel i7 3770K CPU, running under Linux and compiled with the GCC 4.8.2 compiler and -O2 optimization, we were able to measure the execution times shown in Table 3.1, for checking the range $[10^3, 10^7]$. With the exception of the single thread case, where the no-coordination-overheads version #1 wins, in all other cases the last, monitor-based version is superior.

21. Use the `QtConcurrent` functionality to implement a prime number checker. Compare it in terms of speed, efficiency and programming effort to your `QThread`-based attempt of the previous exercise.

**Answer**

The solution involves a `NumberRange` class for representing parts of the number range to scan for prime numbers. When an instance of this class is processed by the static `NumberRange::process` method, the prime numbers in that range are deposited in the designated repository as referenced by the `repo` pointer.

The main method uses the user-supplied desired length of the `NumberRange` instances (variable `PARTLEN`), to populate a vector of them (lines 80-90). This vector is subsequently processed by using the “mapping” functionality of the QtConcurrent namespace (line 92). The `blockingMap` function applies the `NumberRange::process` method on all elements of the input vector, effectively calculating the desired results.

```
1  #include <QtConcurrent/QtConcurrentMap>
2  #include <QMutex>
3  #include <QMutexLocker>
4  #include <math.h>
5  #include <stdlib.h>
6  #include <iostream>
7  #include <vector>
8
9  using namespace std;
10
11 //==========================================
12 // used for storing the results in a thread-safe manner
13 class Repository
14 {
15 private:
16     vector<int> result;
17     QMutex l;
18 public:
19     void store(int i);
20     vector<int> *getResult();
21 };
22 //------------------------------------------
23 void Repository::store(int i)
24 {
25     QMutexLocker ml(&l);
26     result.push_back(i);
27 }
28 //------------------------------------------
29 vector<int> *Repository::getResult()
30 {
31     return &result;
32 }
33 //==========================================
34 class NumberRange
35 {
36 private:
37     int first, last;
38     Repository *repo;
39 public:
40     NumberRange(){}
41     NumberRange(int a, int b, Repository *r) : first(a), last(b),↩
           repo(r){}
42     void setFirst(int f) {first=f;}
43     void setLast(int l) {last=l;}
44     void setRepo(Repository *r) {repo=r;}
45     static void process(NumberRange &);
46 };
47 //------------------------------------------
48 void NumberRange::process(NumberRange &n)
49 {
50     for(int i=n.first; i<=n.last; i+=2)
51     {
52         bool isPrime=true;
53         int j=3;
54         int limit = sqrt(i);
55         while(j<= limit && isPrime)
56         {
57             if(i % j == 0)
58                 isPrime=false;
59             j+=2;
60         }
61         if(isPrime)
62             n.repo->store(i);
63     }
64 }
```

```
//============================================
int main(int argc, char *argv[])
{
    int a, b;
    a=atoi(argv[1]);
    b=atoi(argv[2]);
    int PARTLEN=atoi(argv[3]);

    // make sure a and b are odd
    a = (a % 2 == 0) ? a+1 : a;
    b = (b % 2 == 0) ? b-1 : b;
    PARTLEN = (PARTLEN % 2 == 0) ? PARTLEN : PARTLEN+1; // PARTLEN should be even

    Repository rep;
    vector<NumberRange> data;
    int numParts = (int)ceil((b-a)*1.0/PARTLEN);
    data.resize(numParts);
    int f,l=a-2;
    for(int i=0;i<numParts;i++)
    {
        f=l+2;
        l=f+PARTLEN;
        if(l>b) l=b;
        data.at(i).setFirst(f);
        data.at(i).setLast(l);
        data.at(i).setRepo(&rep);
    }

    QtConcurrent::blockingMap(data, NumberRange::process);

    cout << "Total primes found " << rep.getResult()->size() << endl;
    return 0;
}
```

To run the above program in order to find all the prime numbers in the range $[10^3, 10^8]$, one has to use the following command-line:

```
$ primeChecker_V3 1000 100000000 100000
```

The last parameter concerns the value of `PARTLEN`.

In terms of performance, an identical set of tests as in the previous exercise, results in an average execution time of 0.42 sec, for `PARTLEN = 10000`, which is on par with the monitor-based solution. In terms of development effort, the `QtConcurrent` based solution is marginally shorter.

22. Create a big array of randomly generated 2D coordinates $(x, y)$. Each of the coordinates should be a number in the range [-1000, 1000]. Use appropriate `QtConcurrent` functions to find the points that are in a ring of distances between 100 and 200 from the point of origin. Compare the performance of your solution against a sequential implementation.

    **Answer**

    In the following program a `Point` structure is used to hold randomly generated data. A `vector<Point>` container holds the input data, instead of a `vector<Point*>`, in order to simplify and speedup memory management. Upon resizing the vector container in line 57, all the memory required for holding the `Point` instances is reserved.

    Also, in order to avoid the use of the costly `sqrt` function, the filtering is performed on the squares of the distances.

```cpp
#include <QtConcurrent/QtConcurrentMap>
#include <QtConcurrent/QtConcurrentFilter>
#include <QThreadPool>
#include <QTime>
#include <boost/bind.hpp>
#include <vector>
#include <iostream>
#include <stdlib.h>

using namespace std;

//---------------------------------
struct Point
{
  int x, y;
  int d2;                         // distance square
    Point () {};
};

//---------------------------------
void d2Calc (Point & p)
{
  p.d2 = p.x * p.x + p.y * p.y;
}

//---------------------------------
bool inRing (Point p, int minDist, int maxDist)
{
  int d = p.d2;
  if (d >= minDist && d <= maxDist)
    return true;
  return false;
}

//---------------------------------
int main (int argc, char *argv[])
{
  int numThr;
  int numPoints;
  vector < Point > data;
  double tio, tfull;
  QTime t;

  if (argc < 3)
    {
      cerr << argv[0] << " numThreads numPoints\n";
      exit (1);
    }
  // for timing
  t.start ();

  numThr = atoi (argv[1]);
  numPoints = atoi (argv[2]);

  // initialize data array
  srand (time (0));
  data.resize (numPoints);      // one memory allocation only
  for (int i = 0; i < numPoints; i++)
    {
      data[i].x = rand () % 2000 - 1000;
      data[i].y = rand () % 2000 - 1000;
    }

  tio = t.elapsed () / 1000.0;
  // control the number of threads
  QThreadPool::globalInstance ()->setMaxThreadCount (numThr);

  // calculate distances first
  QtConcurrent::blockingMap (data, d2Calc);

  // filter pointes based on distance next
  QtConcurrent::blockingFilter (data, boost::bind (inRing, _1, ↩
      100 * 100, 200 * 200));

  tfull = t.elapsed () / 1000.0;
  cout << "Total: " << tfull << " Comp.time: " << tfull - tio << ↩
      endl;
  cout << data.size () << endl;
  return 0;
}
```

23. Use the `QtConcurrent` functionality to implement a parallel bucketsort. Does the number of buckets play a significant role in your implementation's performance?.

    **Answer**

    In the following Listing, the `bucketsort` template function implements a variation of the algorithm, whereas (i) the input is scanned and distributed to buckets (lines 43-48), (ii) the buckets are sorted separately via STL's `sort` function (lines 51-54), and (iii) the resulting partial sorted arrays are copied back to the original data repository (lines 57-62).

    The number of buckets matches the number of threads used by `QThreadPool`, and for this reason the number of buckets is invariably connected with the level of concurrency.

    To avoid expensive synchronization logic, such as using mutices to control access to the buckets (STL's `vector` is not thread safe), each thread is responsible for a dedicated bucket. The bucket array address and bucket index for which a thread is responsible for, are passed as parameters to the `filterOp` template function that is invoked by `QtConcurrent::run` in line 45. The minimum value of the input range and the value range per bucket are also passed to enable `filterOp` to determine the bucket that an item belongs to.

    Finally, the sorting is performed via the unary front-end `vecSort` function that is "mapped" on all the buckets. To this end, a `vector` of `vector<T>` pointers is populated in lines 52-53 prior to the application of `QtConcurrent::blockingMap`.

```
1  #include <QtConcurrent/QtConcurrentMap>
2  #include <QtConcurrent/QtConcurrentRun>
3  #include <QFuture>
4  #include <QThreadPool>
5  #include <QTime>
6  #include <boost/bind.hpp>
7  #include <vector>
8  #include <algorithm>
9  #include <iostream>
10 #include <stdlib.h>
11
12 using namespace std;
13
14 const int MINV = -1000;
15 const int MAXV = 1000;
16
17 //-------------------------------------
18 template <typename T> inline void filterOp(vector<T> *v, int ↩
       targetBucketIdx, vector<T> *b, T m, T bWidth)
19 {
20     for(int i=0;i< v->size();i++)
21     {
22         T tmp = v->at(i);
23         int bucketIdx = (tmp - m) / bWidth;
24         if(bucketIdx == targetBucketIdx)
25             b->push_back(tmp);
26     }
```

```
}

//———————————————————————————————————————
template <typename T> inline void vecSort(vector<T> *v)
{
    sort(v->begin(), v->end());
}

//———————————————————————————————————————
template <typename T> void bucketSort(vector<T> &data, const T &↩
    minV, const T &maxV)
{
    int numBuckets = QThreadPool::globalInstance()->↩
        maxThreadCount(); //match buckets with the number of ↩
        threads
    vector<T> bucket[numBuckets];
    T bwidth = (maxV - minV)/numBuckets;  // range per bucket

    // launch as many requests as the number of buckets/threads
    QFuture<void> f[numBuckets];
    for(int i=0;i<numBuckets;i++)
        f[i] = QtConcurrent::run(boost::bind(filterOp<T>, &data, ↩
            i, bucket + i, minV, bwidth));

    for(int i=0;i<numBuckets;i++)
        f[i].waitForFinished();

    // sort each of the buckets separately
    vector< vector<T> *> vv;
    for(int i=0;i<numBuckets;i++)
        vv.push_back(&(bucket[i]));
    QtConcurrent::blockingMap(vv, vecSort<T>);

    // copy back to original vector
    int loc=0;
    for(int i=0;i<numBuckets;i++)
    {
        for(int j=0;j<bucket[i].size();j++)
            data[loc++] = bucket[i].at(j);
    }
}

//———————————————————————————————————————
int main (int argc, char *argv[])
{
    int numThr;
    int numItems;
    vector < int > data;
    double tio, tfull;
    QTime t;

    if (argc < 3)
    {
        cerr << argv[0] << " numThreads numItems\n";
        exit (1);
    }
    // for timing
    t.start ();

    numThr = atoi (argv[1]);
    numItems = atoi (argv[2]);

    // initialize data array
    srand (time (0));
    data.resize (numItems);      // one memory allocation only
    for (int i = 0; i < numItems; i++)
    {
        data[i] = (rand () % (MAXV-MINV)) + MINV;
    }

    tio = t.elapsed () / 1000.0;
    // control the number of threads
    QThreadPool::globalInstance ()->setMaxThreadCount (numThr);

    bucketSort<int>(data, MINV, MAXV);

    tfull = t.elapsed() / 1000.0;
    cout << "Total: " << tfull << " Comp.time: " << tfull - tio ↩
        << endl;
    return 0;
}
```

# Chapter 4

# Shared-memory programming : OpenMP

## Exercises

1. Modify the program in Listing 4.1, so that `printf` is used instead of `cout`. Do you see any difference in the output compared to the one reported in Section 4.2? Can you explain it?

   **Answer**

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

using namespace std;

int main (int argc, char **argv)
{
  int numThr = atoi (argv[1]);
#pragma omp parallel num_threads(numThr)
  printf("Hello from thread %i\n", omp_get_thread_num ());

  return 0;
}
```

   The output does not appear mangled, as the `printf` strings are build and output as a unit, while the `cout` object goes through several method invocations.

2. In the matrix multiplication example of Section 4.4.2 we could get three perfectly nested loops, if we initialize the `C` matrix outside:

```
  double A[K][L];
  double B[L][M];
  double C[K][M];
. . .
#pragma omp parallel for collapse(3)
  for (int i = 0; i < K; i++)
    for (int j = 0; j < M; j++)
      for (int k = 0; k < L; k++)
        C[i][j] += A[i][k] * B[k][j];
```

   Is the above code correct? If not, what kind of modification is required for fixing it?

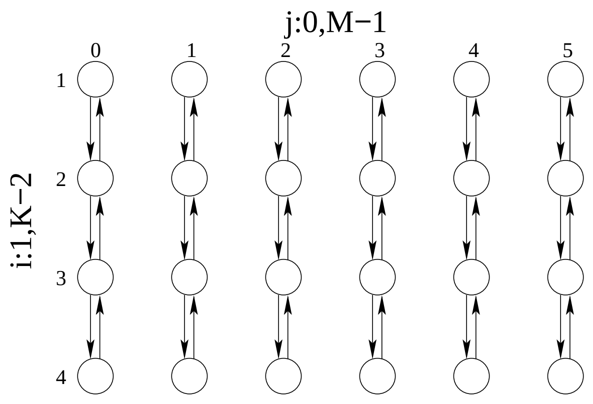


Figure 4.1: Dependency graph for Exercise 3.

**Answer**

The code is not correct, because for the inner loop `C[i][j]` is effectively a reduction variable. To eliminate the race condition, a critical section is needed:

```
#pragma omp parallel for collapse(3)
  for (int i = 0; i < K; i++)
    for (int j = 0; j < M; j++)
      for (int k = 0; k < L; k++)
#pragma omp critical
        C[i][j] += A[i][k] * B[k][j];
```

In terms of performance, this will effectively make all threads execute sequentially, because unnamed `critical` constructs use the same mutex.

A better alternative is to make sure that each execution of the inner loop is assigned to a single thread, by using a `schedule` clause, which is equivalent to converting the `collapse(3)` clause to `collapse(2)`:

```
#pragma omp parallel for collapse(3) schedule(static, L)
  for (int i = 0; i < K; i++)
    for (int j = 0; j < M; j++)
      for (int k = 0; k < L; k++)
        C[i][j] += A[i][k] * B[k][j];
```

3. Draw the iteration space dependency graph for the following program:

```
  for (int i = 1; i < K-1; i++)
    for (int j = 0; j < M; j++)
        a[i][j] = a[i-1][j] + a[i+1][j];
```

What kind of dependencies exist? How can you eliminate them?

**Answer**

The ISDG is shown in Figure 4.1.

We have a flow dependence from `a[i-1][j]` and an anti-dependence for a[i+1][j]. The anti-dependence can be eliminated by creating a copy `a2` of the original matrix `a`. The flow dependence can be honored (not removed) by parallelizing only the inner loop, resulting in the following code:

```
  for (int i = 0; i < K-1; i++)
    for (int j = 0; j < M; j++)
        a2[i][j] = a[i+1][j];

  for (int i = 1; i < K-1; i++)
#pragma omp parallel for
    for (int j = 0; j < M; j++)
        a[i][j] = a[i-1][j] + a2[i][j];
```

4. Create a C++ program for visualizing the thread iteration assignment performed by a `parallel for` directive, for different `schedule` schemes. Your program should have a per-thread vector that accumulates the loop control variable values assigned to each thread. Use this program, and appropriate `schedule` settings, to experiment with multiple schemes, without recompiling your program. This is a sample of the output you should be able to get for a loop of 100 iterations:

```
$ export OMP_SCHEDULE="static,5"
$ ./solution
Thread 0 : 0 1 2 3 4 40 41 42 43 44 80 81 82 83 84
Thread 1 : 5 6 7 8 9 45 46 47 48 49 85 86 87 88 89
Thread 2 : 10 11 12 13 14 50 51 52 53 54 90 91 92 93 94
Thread 3 : 15 16 17 18 19 55 56 57 58 59 95 96 97 98 99
Thread 4 : 20 21 22 23 24 60 61 62 63 64
Thread 5 : 25 26 27 28 29 65 66 67 68 69
Thread 6 : 30 31 32 33 34 70 71 72 73 74
Thread 7 : 35 36 37 38 39 75 76 77 78 79
```

**Answer**

The solution is based on having a vector per thread (line 11), to store the values of the loop control variable (line 18):

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <vector>
4  #include <omp.h>
5
6  using namespace std;
7
8  int main ()
9  {
10   int N = omp_get_max_threads ();
11   vector < int >*store = new vector < int >[N];
12
13
14 #pragma omp parallel for schedule( runtime )
15   for (int k = 0; k < 100; k++)
16     {
17       int idx = omp_get_thread_num ();
18       store[idx].push_back (k);
19     }
20
21   //output
22   for (int i = 0; i < N; i++)
23     {
24       cout << "Thread " << i << " : ";
25       for (int j = 0; j < store[i].size (); j++)
26         cout << store[i][j] << " ";
27       cout << endl;
28     }
29
30   return 0;
31 }
```

5. In Listing 4.19 we examined the issue of processing a linked-list concurrently. Is there a way to improve this program for a doubly-linked list? Write the corresponding program.

**Answer**

In the following code, the list is traversed in two opposite directions, allowing the generation of two tasks per iteration of the `while` loop of lines 61-75.

```
1  #include <iostream>
2  #include <math.h>
3  #include <stdlib.h>
4  #include <omp.h>
5  #include <unistd.h>
6
7  using namespace std;
8
9  // template structure for a list's node
10 template < class T > struct Node
11 {
12   T info;
13   Node *next;
14   Node *prev;
15 };
16 //-------------------------------------------
17 // Appends a value at the end of a list pointed by the head *h and tail *t
18 template < class T > void append (int v, Node < T > **h, Node < T > **t)
19 {
20   Node < T > *tmp = new Node < T > ();
21   tmp->info = v;
22   tmp->next = NULL;
23   tmp->prev = *t;
24
25   if (*t == NULL)
26     {
27       *h = tmp;
28       *t = tmp;
29     }
30   else
31     {
32       (*t)->next = tmp;
33       *t = tmp;
34     }
35 }
36
37 //-------------------------------------------
38 // function stub for processing a node's data
39 template < class T > void process (Node < T > *p)
40 {
41 #pragma omp critical
42   cout << p->info << " by thread " << omp_get_thread_num () << endl;
43 }
44
45 //-------------------------------------------
46 int main (int argc, char *argv[])
47 {
48   // build a sample list
49   int N = atoi (argv[1]);
50   Node < int >*head = NULL;
51   Node < int >*tail = NULL;
52   for (int i = 0; i < N; i++)
53     append (i, &head, &tail);
54
55 #pragma omp parallel
56   {
57 #pragma omp single
58     {
59       Node < int >*tmp1 = head;
60       Node < int >*tmp2 = tail;
61       while (tmp1 != NULL && tmp1 != tmp2)
62         {
63 #pragma omp task
64           process (tmp1);
65 #pragma omp task
```

```
          process (tmp2);

          tmp1 = tmp1->next;
          if (tmp1 == tmp2)      // even-sized list termination check
            {
              tmp1 = NULL;
              break;
            }
          tmp2 = tmp2->prev;
        }
      // process the last node, in the middle of an odd-sized list
      if (tmp1 != NULL)
        process (tmp1);
    }
  }

  return 0;
}
```

6. Write a program for traversing and processing the elements of a binary tree in parallel, using a pre-order, in-order or post-order traversal. Use the `task` construct to that effect.

   **Answer**

   In-order traversal cannot be parallelized as in enforces a total order between the tree nodes.

```
//--------------------------
template < typename T > struct TreeNode
{
  T value;
    TreeNode < T > *left;
    TreeNode < T > *right;
};
//--------------------------
// empty placeholder for a processing function
template < typename T > void processStub (T n)
{
}

//--------------------------

template < typename T > void preOrder (TreeNode < T > *n, void *process (T n))
{
  if (n == NULL)
    return;

  process (n->value);
#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task
      preOrder (n->left, process);

      // keep using this thread for the other child
      preOrder (n->right, process);
#pragma omp taskwait
    }
  }
}

//--------------------------

template < typename T > void postOrder (TreeNode < T > *n, void *process (T n))
{
  if (n == NULL)
```

```
    return;

#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task
      postOrder (n->left, process);

      // keep using this thread for the other child
      postOrder (n->right, process);
#pragma omp taskwait
    }
  }
  process (n->value);
}

//--------------------------

// inOrder cannot be parallelized as in enforces a total order between the tree nodes.
// postOrder and preOrder only have a partial ordering
template < typename T > void inOrder (TreeNode < T > *n, void *process (T n))
{
  if (n == NULL)
    return;

  inOrder (n->left, process);
  process (n->value);
  inOrder (n->right, process);

}
```

7. Modify the program of the previous exercise, so that *undeferred tasks* are generated after the traversal has moved beyond the fifth level of the tree (assume that the root sits at level 0).

   **Answer**

   The required modifications relative to the answer of the previous question, are shown below. The solution is based on the introduction of a function parameter for counting the depth of the recursion/depth of the tree nodes examined. The default value of "0" means that the caller does not need to be aware of this parameter.

```
template < typename T > void preOrder (TreeNode < T > *n, void *process (T n), int level=0)
{
  if (n == NULL)
    return;

  process (n->value);
#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task if(level>=5)
      preOrder (n->left, process, level+1);

      // keep using this thread for the other child
      preOrder (n->right, process, level+1);
#pragma omp taskwait
    }
  }
}

//--------------------------

template < typename T > void postOrder (TreeNode < T > *n, void *process (T n), int level=0)
```

```
{
  if (n == NULL)
    return;

#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task if(level>=5)
      postOrder (n->left, process, level+1);

      // keep using this thread for the other child
      postOrder (n->right, process, level+1);
#pragma omp taskwait
    }
  }
  process (n->value);
}
```

8. Modify the Fibonacci sequence-calculating program of Listing 4.20, so that it counts the number of child tasks generated.

   **Answer**

   Use of an `atomic` pragma is required for properly incrementing the shared counter. Only changes relative to Listing 4.20 are shown.

```
int numTasks = 0;

int fib (int i)
{
  int t1, t2;
  if (i == 0 || i == 1)
    return 1;
  else
    {
#pragma omp task shared(t1) if(i>25) mergeable
      {
#pragma omp atomic
        numTasks++;
        t1 = fib (i - 1);

      }
// keep thread for calling function again
      t2 = fib (i - 2);
#pragma omp taskwait
      return t1 + t2;
    }
}
```

9. Use the `task` directive to create a solution to the single producer, multiple consumers problem, with a variable number of consumers, as specified in the command-line.

   **Answer**

   The following code solves the integration example that is presented in Section 4.5.1.1 and solved with a fixed number of consumers in Listing 4.18. To minimize clutter, we show only the `main` function that generates the consumer threads (via a `for` loop in lines 21-25) and executes the producer thread code (lines 28-52).

```
1 int main (int argc, char **argv)
2 {
3   if (argc == 1)
4     {
5       cerr << "Usage " << argv[0] << " #threads #jobs\n";
6       exit (1);
```

```
    }
  int N = atoi (argv[1]);
  int J = atoi (argv[2]);

  Slice *buffer = new Slice[BUFFSIZE];
  int in = 0, out = 0;
  QSemaphore avail, buffSlots (BUFFSIZE);
  QMutex l, integLock;
  double integral = 0;
#pragma omp parallel default(none) shared(buffer, in, out, avail,↩
     buffSlots, l, integLock, integral, J, N, cout) num_threads(↩
    N+1)
  {
#pragma omp single
    {
// consumers' part
      for (int i = 0; i < N; i++)
        {
#pragma omp task
          integrCalc (buffer, buffSlots, avail, l, out, integLock↩
              , integral);
        }

// producer thread, responsible for handing out 'jobs'
      double divLen = (UPPERLIMIT - LOWERLIMIT) / J;
      double st, end = LOWERLIMIT;
      for (int i = 0; i < J; i++)
        {
          st = end;
          end += divLen;
          if (i == J - 1)
            end = UPPERLIMIT;

          buffSlots.acquire ();
          buffer[in].start = st;
          buffer[in].end = end;
          buffer[in].divisions = 1000;
          in = (in + 1) % BUFFSIZE;
          avail.release ();
        }

      // put termination sentinels in buffer
      for (int i = 0; i < N; i++)
        {
          buffSlots.acquire ();
          buffer[in].divisions = 0;
          in = (in + 1) % BUFFSIZE;
          avail.release ();
        }

// wait for all threads to finish
#pragma omp taskwait
    }
  }

  cout << "Result is : " << integral << endl;
  delete[] buffer;

  return 0;
}
```

10. Use the `task` directive and its `depend` clause, to model the dependency graph of Figure 4.12. You can use stub functions to represent each of the tasks in the figure.

    **Answer**

```
#include <iostream>
#include <stdlib.h>
#include <omp.h>
#include <unistd.h>

```

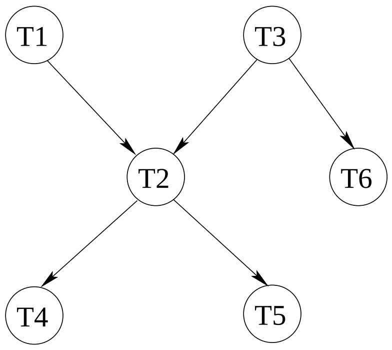


Figure 4.2: A dependency graph consisting of six tasks.

```
using namespace std;

int main (int argc, char **argv)
{
  int x, y, z;

#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task  shared(x) depend(out : x)
      {  // T1
        cout << "T1\n";
      }

#pragma omp task shared(x) depend(out : y, z)
      {  // T3
        cout << "T3\n";
      }

#pragma omp task  shared(x) depend(inout : x, y)
      {  // T2
        sleep (1);
        cout << "T2\n";
      }

#pragma omp task shared(x) depend(in : y)
      {  // T4
        cout << "T4\n";
      }

#pragma omp task  shared(x) depend(in : y)
      {  // T5
    sleep(1);
        cout << "T5\n";
      }

#pragma omp task shared(x) depend(in : z)
      {  // T6
        cout << "T6\n";
      }

#pragma omp taskwait
    }
  }

return 0;
}
```

11. Finding the odd integers in a array can be accomplished by the following

OpenMP code:

```
int data[N];
int oddCount=0;
#pragma omp parallel for
 for (int i = 0; i < N; i++)
      if( data[i] % 2 )
#pragma omp atomic
         oddCount ++;
```

Modify the above code, so that there is no need for a `critical` or `atomic` directive, by introducing a counter array, with one element per thread. Compare the performance of your version with and without cache false sharing. If we were to use a reduction variable, what false-sharing elimination technique, does this correspond to?

**Answer**

The following listing contains four different versions of the code snippet in question, encapsulated in appropriately named functions:

```
// Original code wih atomic
int count_atomic (int *data, int N)
{
  int oddCount = 0;
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    if (data[i] % 2)
#pragma omp atomic
      oddCount++;
  return oddCount;
}

//----------------------------------------------
// Array of counters, but there is false sharing
int count_false_sharing (int *data, int N)
{
  int numThr = omp_get_max_threads ();
  int oddCount[numThr];
  memset (oddCount, 0, numThr * sizeof (int));
  int totalCount = 0;
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    {
      if (data[i] % 2)
        oddCount[omp_get_thread_num ()]++;
    }

  for (int i = 0; i < numThr; i++)
    totalCount += oddCount[i];
  return totalCount;
}

//----------------------------------------------
// Eliminates false sharing with padding
int count_without_false_sharing (int *data, int N)
{
  int numThr = omp_get_max_threads ();
  int oddCount[numThr * 8];
  memset (oddCount, 0, numThr * 8 * sizeof (int));
  int totalCount = 0;
#pragma omp parallel for
  for (int i = 0; i < N; i++)
    {
      if (data[i] % 2)
        oddCount[omp_get_thread_num () * 8]++;
    }

  for (int i = 0; i < numThr; i++)
    totalCount += oddCount[i * 8];
  return totalCount;
}
```

```

//---------------------------------------------
int count_with_reduction (int *data, int N)
{
  int oddCount = 0;
#pragma omp parallel for reduction(+:oddCount)
  for (int i = 0; i < N; i++)
    {
      if (data[i] % 2)
        oddCount++;
    }
  return oddCount;
}
```

The use of a reduction variable is equivalent to using private variables for eliminating false sharing.

12. Quicksort rightfully holds the place of the top-performer amongst generic sorting algorithms. It's design can be considered a reflection of mergesort, although both algorithms employ a divide-and-conquer design. Write a sequential implementation of quicksort and proceed to parallelize it using OpenMP constructs. Measure the speedup that can be achieved, and compare it to the performance that can be obtained from the bottom-up mergesort of Section 4.8.1.

    **Answer**

    The solution involves the use of task constructs in lines and . The code incorporates several quicksort optimizations such as median-of-three pivot element selection (lines 54-62) and a switch to insertion sort once the size of the data fall below a fixed threshold (lines 92-96). To reduce the generation of tasks, a similar heuristic to the one described in Section 4.8.2 is used (`_thresh_` variable in line 121).

    In terms of performance, the program more closely matches the top-down mergesort, failing to fully exploit a multicore CPU because of its recursive nature. The bottom-up mergesort was about twice as fast as the quicksort on an i7 3770K CPU.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <omp.h>
5  #include <QTime>
6  #include <iostream>
7
8  using namespace std;
9
10 const int INSERTION_THRESHOLD=50;
11 const int maxTasks=256;
12 int _thresh_; // used for the if clauses
13
14 //**************************************************
15 void insertionSort(int *data, int N)
16 {
17     for (int i = 1; i < N; i++)
18     {
19         int tmp = data[i];
20         int loc=i-1;
21         while(loc >=0 && data[loc] > tmp)
22         {
23             data[loc+1] = data[loc];
24             loc--;
25         }
26         data[loc+1] = tmp;
27     }
```

```
}

//***********************************************
void numberGen (int N, int max, int *store)
{
    int i;
    srand (time (0));
    for (i = 0; i < N; i++)
        store[i] = rand () % max;
}

//***********************************************
void swap (int *data, int x, int y)
{
    int temp = data[x];
    data[x] = data[y];
    data[y] = temp;
}

//***********************************************
int partition (int *data, int N)
{
    int i = 0, j = N;

    // median of 3
    int a=data[0], b=data[N >> 1], c=data[N-1];
    if((a > b && b >c) || (c > b && b >a))
    {
        swap(data, 0, N>>1);
    }
    else if((a > c && c >b) || (b > c && c>a))
    {
        swap(data, 0, N-1);
    }

    int pivot = data[0];

    do
    {
        do
        {
            i++;
        }
        while (pivot > data[i] && i < N);

        do
        {
            j--;
        }
        while (pivot < data[j] && j > 0);
        swap (data, i, j);
    }
    while (i < j);

    swap (data, i, j);

    swap (data, 0, j);
    return j;
}

//***********************************************
void QSort (int *data, int N)
{
    if (N <= 1)
        return;
    else if(N<INSERTION_THRESHOLD)
    {
        insertionSort(data, N);
        return;
    }

    int pivotPos = partition (data, N);
#pragma omp parallel
    {
```

```
#pragma omp single
        {
#pragma omp task if(N > _thresh_ ) mergeable
            QSort (data, pivotPos);
#pragma omp task  if(N > _thresh_ ) mergeable
            QSort (&(data[pivotPos + 1]), N - pivotPos - 1);
#pragma omp taskwait
        }
    }
}

//-------------------------------------------------------------
int main (int argc, char *argv[])
{
    if (argc == 1)
    {
        fprintf (stderr, "%s N\n", argv[0]);
        exit (0);
    }
    int N = atoi (argv[1]);
    _thresh_ = 2.0 * N / (maxTasks + 1);

    int *data = new int[N];
    numberGen (N, 1000, data);
    QTime t;
    t.start();

    QSort (data, N);

    cout << "Sorted in " << t.elapsed()/1000.0 << " sec\n";

    delete [] data;
    return 0;
}
```

# Chapter 5

# Distributed memory programming

## Exercises

1. Write a MPMD version of the "Hello World" program of Figure 5.2, in effect eliminating the `if/else` structure around which the program is built. You may use the C or C++ bindings.

   **Answer**

   Two source code files are needed:

```
// File : hello_node0.c
#include<mpi.h>
#include<string.h>
#include<stdio.h>
#define MESSTAG 0
#define MAXLEN 100
int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, num, i;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  char mess[] = "Hello World";
  for (i = 1; i < num; i++)
    MPI_Send (mess, strlen (mess) + 1, MPI_CHAR, i, MESSTAG, ↩
        MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

```
// File : hello_node_i.c
#include<mpi.h>
#include<string.h>
#include<stdio.h>
#define MESSTAG 0
#define MAXLEN 100
int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, num, i;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  char mess[MAXLEN];
  MPI_Status status;
  MPI_Recv (mess, MAXLEN, MPI_CHAR, 0, MESSTAG, MPI_COMM_WORLD, &↩
      status);
  printf ("%i received %s\n", rank, mess);

  MPI_Finalize ();
  return 0;
}
```

The program can run with the following appfile:

```
# File :  hello_app
-host localhost -np 1 hello_node0
-host localhost -np 3 hello_node_i
```

and via the command sequence:

```
$ mpicc hello_node0.c -o hello_node0
$ mpicc hello_node_i.c -o hello_node_i
$ mpirun -app hello_app
```

2. Write a SPMD version of the two programs shown in Listing 5.5.

   **Answer**

   The only requirement is the inclusion of an `if-else` control logic.

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXLEN 100
char *greetings[] = { "Hello", "Hi", "Awaiting your command" };

char buff[MAXLEN];

int main (int argc, char **argv)
{
  MPI_Status st;
  int procNum;
  int rank;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &procNum);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  if (rank == 0)
    {
      while (--procNum)
        {
          MPI_Recv (buff, MAXLEN, MPI_CHAR, MPI_ANY_SOURCE, ↩
              MPI_ANY_TAG, MPI_COMM_WORLD, &st);
          int aux;
          MPI_Get_count (&st, MPI_CHAR, &aux);
          buff[aux] = 0;
          printf ("%s\n", buff);
        }
    }
  else
    {
      srand (time (0));
      int grID = rand () % 3;
      sprintf (buff, "Node %i says %s", rank, greetings[grID]);
      MPI_Send (buff, strlen (buff), MPI_CHAR, 0, 0, ↩
          MPI_COMM_WORLD);
    }

  MPI_Finalize ();
}
```

3. Modify the program shown in Listing 5.5 so that the master node prints out a list of the processes IDs for which the message has not been read yet. Your output should be similar to:

```
$ mpirun -np 1 master : -np 3 worker
Node 2 says Hi. Awaiting nodes : 1 3
Node 3 says Hi. Awaiting nodes : 1
Node 1 says Hi. Awaiting nodes :
```

**Answer**

An STL `set<int>` container can be used to hold the ID's of the processes still pending:

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <set>
#include <iostream>

using namespace std;

#define MAXLEN 100
char buff[MAXLEN];

int main (int argc, char **argv)
{
  MPI_Status st;
  int procNum;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &procNum);
  set<int> leftOver;
  for(int i=1;i<procNum;i++)
    leftOver.insert(i);

  while (--procNum)
    {
      MPI_Recv (buff, MAXLEN, MPI_CHAR, MPI_ANY_SOURCE, ↩
          MPI_ANY_TAG,
        MPI_COMM_WORLD, &st);
      int aux;
      MPI_Get_count (&st, MPI_CHAR, &aux);
      leftOver.erase(st.MPI_SOURCE);

      buff[aux] = 0;
      printf ("%s. Awaiting nodes", buff);
      for (set<int>::iterator it=leftOver.begin(); it!=leftOver.↩
          end(); ++it)
         cout << ' ' << *it;
      cout << '\n';

    }
  MPI_Finalize ();
}
```

4. Create a model of the characteristics of the communication link joining two processes running on two different machines, i.e. calculate the start-up latency and communication rate, by implementing and testing a **ping-pong** benchmark program. A ping-pong program measures the time elapsed between sending a message, having it bounce at its destination, and receiving it back at its origin. By varying the message size, you can use statistical methods (least-squares) to estimate the start-up latency and rate as the intercept and slope respectively, of the line fitted to the experimental data.

**Answer**

The following code will calculate the average communication time over `REP` number of tests between two processes, and for message sizes ranging from 0 to `MAX_MESG` in increments of 1000.

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

using namespace std;

#define MESG_TAG 0
#define END_TAG 1
#define MAX_MESG 1000000

const int REP = 10;

int main (int argc, char *argv[])
{
  int size, rank;
  int namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int mesg_size;
  int tag;
  char *buffer;
  double start_time, end_time;
  MPI_Status status;

  buffer = new char[MAX_MESG];

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (processor_name, &namelen);

  printf ("Process %d of %d on %s\n", rank, size, processor_name);
  if (size < 2)
    {
      printf ("Need more than 1 processor to run\n");
      exit (1);
    }

  if (rank == 0)
    {
      for (int mesg_size = 0; mesg_size <= MAX_MESG; mesg_size += 1000)
        {
          start_time = MPI_Wtime ();
          for (int i = 0; i < REP; i++)
            {
              MPI_Send (buffer, mesg_size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
              MPI_Recv (buffer, MAX_MESG, MPI_CHAR, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            }

          end_time = MPI_Wtime ();
          printf ("%i %lf\n", mesg_size, (end_time - start_time) / 2 / REP);
        }

      tag = END_TAG;
      MPI_Send (buffer, 0, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
    }
  else
    {
      while (1)
        {
          MPI_Recv (buffer, MAX_MESG, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          if (status.MPI_TAG == END_TAG)
            break;
          MPI_Get_count (&status, MPI_CHAR, &mesg_size);
          MPI_Send (buffer, mesg_size, MPI_CHAR, 0, tag,
                MPI_COMM_WORLD);
        }
    }

  delete[] buffer;
  MPI_Finalize ();
  return 0;
}
```

5. How would we need to modify the broadcasting program of Listing 5.9, if the source of the message was an arbitrary process, and not the one with rank 0?

   **Answer**

   The solution follows the guidelines of Listing 5.9, i.e. treating the processes as nodes in a hypercube, and proceeding to propagate the message along the hypercube's dimension one-by-one. However, the source-destination process pairings has to be done using an alternative approach. The key to the pairing calculation is the communication phase during which a node receives data. This is equal to the distance of a process/node from the root on the virtual hypercube, which can be calculated by counting the number of bits in the `rank XOR root` expression (variable `recvPhase`, line 35).

   The sender node can be found by reversing the bit of a node's rank in the `recvPhase - 1` position (line 51). Once a node receives the message, it can start to send it to all the nodes which are adjacent to it (ranks different by a single bit). The ranks of the receivers can be determined by reversing the bits in a node's rank from position `recvPhase` and onwards (lines 58-64).

```
1   #include<mpi.h>
2   #include<string.h>
3   #include<stdio.h>
4   #define MESSTAG 0
5
6   //******************************************
7   // Returns the number of set bits in its argument
8   int bitCount (int i)
9   {
10    int count = 0;
11    while (i != 0)
12      {
13        int j = i;
14        i >>= 1;
15        if (i != j)
16          count++;
17      }
18    return count;
19  }
20
21  //******************************************
22
23  int main (int argc, char **argv)
24  {
25    MPI_Init (&argc, &argv);
26
27    int rank, num, i;
28    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
29    MPI_Comm_size (MPI_COMM_WORLD, &num);
30    MPI_Status status;
31    int root = atoi (argv[1]);
32    double data = rank * 1.0; // something simple to enable ↩
          verification of correct receipt
33
```

```
  printf ("%i will be served during phase %i\n", rank, bitCount (rank ^ root));
  int recvPhase = bitCount (rank ^ root);
  if (rank == root)
    {
      int phase = 0;
      int destID = rank ^ (1 << phase);
      while (destID < num)
        {                          // a subset of nodes gets a message
          printf ("#%i sending to %i\n", rank, destID);
          MPI_Send (&data, 1, MPI_DOUBLE, destID, MESSTAG, MPI_COMM_WORLD);
          phase++;
          destID = rank ^ (1 << phase);
        }
    }
  else
    {
      int srcID;
      srcID = rank ^ (1 << (recvPhase - 1));
      printf ("SRC of #%i : %i\n", rank, srcID);

      MPI_Recv (&data, 1, MPI_DOUBLE, srcID, MESSTAG, MPI_COMM_WORLD, &status);

      // calculate the ID of the node that will receive a copy of the message
      int destID = rank ^ (1 << recvPhase);
      while (destID < num)
        {
          printf ("#%i sendin to %i\n", rank, destID);
          MPI_Send (&data, 1, MPI_DOUBLE, destID, MESSTAG, MPI_COMM_WORLD);
          recvPhase++;
          destID = rank ^ (1 << recvPhase);
        }
    }
  printf ("Node #%i has %lf\n", rank, data);
  MPI_Finalize ();
  return 0;
}
```

6. Assuming that the execution platform of your program, consists of 4 machines with identical architecture but different CPU clocks: one with 4GHz, one with 3 GHz and 2 with 2 GHz. How should you split the matrix `A` used in the example of Section 5.11.1, in order to solve the matrix-vector product problem in the smallest possible time?

   **Answer**

   The heterogeneity of the execution platform calls for an uneven partitioning of the problem's data, i.e. we should assign a bigger portion of the problem to the faster nodes. If we assume that the computing speed of the machines is linearly dependent on their clock, and we ignore communication overheads, then if $T$ is the time to complete the multiplication on a 1GHz machine, then on a $C$ times faster-clock machine we would need time $\frac{T}{C}$. If each node $i$ were assigned $part_i$ percent of the $M$ rows of the input matrix, then it would need time $T_i = part_i \frac{T}{C_i}$ to complete its part of the computation.

   It can be easily proven by contradiction, that the overall execution time is minimized if $T_i = T_j$ for $\forall i \neq j$, which translates to:

$$T_i = T_j \Rightarrow part_i \frac{T}{C_i} = part_j \frac{T}{C_j} \Rightarrow part_i = part_j \frac{C_i}{C_j} \tag{5.1}$$

As the sum of all parts should be equal to 1, we can derive the value of $part_0$ and subsequently from Eq. 5.1 all the other parts:

$$\sum_{\forall i} part_i = 1 \Rightarrow part_0 \sum_{\forall i} \frac{C_i}{C_0} = 1 \Rightarrow part_0 = (\sum_{\forall i} \frac{C_i}{C_0})^{-1} \tag{5.2}$$

From the problem description we have $C_0 = 4GHz$, $C_1 = 3GHz$, $C_2 = 2GHz$ and $C_3 = 2GHz$, which when substituted in Eq. 5.1 and 5.2 return:

$$part_0 = (\frac{4}{4} + \frac{3}{4} + \frac{2}{4} + \frac{2}{4})^{-1} = \frac{4}{11}$$

$$part_1 = \frac{4}{11}\frac{3}{4} = \frac{3}{11}$$

$$part_2 = part_3 = \frac{4}{11}\frac{2}{4} = \frac{2}{11}$$

The modified code for setting up the scattering of the matrix to the four nodes, is shown below:

```
double part[] = { 4.0 / 11, 3.0 / 11, 2.0 / 11, 2.0 / 11 };

if (rank == 0)
  {
    int displs[N];
    int sendcnts[N];
    for (int i = 0; i < N; i++)
      {
        int rowsForProcess = M * part[i];
        sendcnts[i] = M * rowsForProcess;
        displs[i] = (i == 0) ? 0 : displs[i - 1] + sendcnts[i -
             1];
        if (i == N - 1)
          sendcnts[i] = M * M - displs[i];
      }

    MPI_Scatterv (A, sendcnts, displs, MPI_DOUBLE, MPI_IN_PLACE
         , 0, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

7. Write a program that performs gathering as efficiently as possible, using point-to-point communications, i.e. the equivalent of `MPI_Gather`. What is the *time complexity* of your algorithm?

**Answer**

The answer is very closely related to the solution of Exercise 5.5. The major thing that needs to be altered, is the reversal of data flow, i.e. replacing `MPI_Send` with `MPI_Recv` and vice-versa. The XOR operator can be still used to calculate the destination and source nodes in the communications.

The other change stems from the need to accumulate the data that are collected. Because the order of collection does not necessarily (and typically does not) match the node ranking, all the parts that are accumulated are prefixed in the same buffer with the rank of the node that contributed them. In this fashion, the designated root of the gathering operation, can reshuffle the parts and produce an outcome compatible with what `MPI_Gather` would produce.

The following program requires as a command-line parameter, the rank of the process/node that will gather the data.

```
#include<mpi.h>
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define MESSTAG 0
#define MESSSIZE 10

//*****************************************
// Returns the number of set bits in its argument
int bitCount (int i)
{
  int count = 0;
  while (i != 0)
    {
      int j = i;
      i >>= 1;
      if (i != j)
        count++;
    }
  return count;
}

//*****************************************

int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, num, i;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Status status;
  int root = atoi (argv[1]);
  int data[MESSSIZE];
  for (int i = 0; i < MESSSIZE; i++)
    data[i] = rank * 1.0;        // something simple to enable verification of correct receipt

  int totalPhases = ceil (log2 (num)); // total collection phases

  int sendPhase = bitCount (rank ^ root); // phase when a node forwards its part of the data
  printf ("%i will sent to root during phase %i\n", rank, sendPhase);

  // allocate the temp. buffer for holding the data to be communicated towards the "root"
  int buffSpace = (MESSSIZE + 1) << (totalPhases - sendPhase);
  if (rank == root)
    buffSpace = num * (MESSSIZE + 1);
  int *commBuff = new int[buffSpace];
  int inCommBuff = MESSSIZE + 1;
  // setup local part of the comm data and the relevant "header"
  commBuff[0] = rank;
  memcpy (commBuff + 1, data, sizeof (int) * MESSSIZE);
  printf ("#%i has BUFF %i\n", rank, buffSpace);

  if (rank == root)
    {
      int phase = totalPhases - 1;
      int srcID = rank ^ (1 << phase);
      while (phase >= 0)
        {
          if (srcID < num)
            {
              printf ("#%i recv from %i  SPACE %i\n", rank, srcID, buffSpace - inCommBuff);
              MPI_Recv (commBuff + inCommBuff, buffSpace - inCommBuff, MPI_INT, srcID, MESSTAG, MPI_COMM_WORLD, &status);
              int recvd;
              MPI_Get_count (&status, MPI_INT, &recvd);
              inCommBuff += recvd;
            }
          phase--;
          srcID = rank ^ (1 << phase);
        }

      // now re-arrange the collected data
      int gatheredData[MESSSIZE * num];
      int pos = 0;
      for (int i = 0; i < num; i++)
        {
          int destPartIdx = commBuff[pos++];
          for (int j = 0; j < MESSSIZE; j++)
            gatheredData[j + destPartIdx * MESSSIZE] = commBuff[↩
                pos++];
        }

      // print-out for verification purposes
      for (int i = 0; i < MESSSIZE * num; i++)
        printf ("%i ", gatheredData[i]);
      printf ("\n");
    }
  else
    {
      int phase = totalPhases;
      // calculate the ID of the node that will send its part of ↩
          the data to this node
      int srcID = rank ^ (1 << (phase - 1));
      // collect from other nodes first
      while (phase > sendPhase)
        {
          if (srcID < num)
            {
              printf ("#%i recv from %i\n", rank, srcID);
              MPI_Recv (commBuff + inCommBuff, buffSpace - ↩
                  inCommBuff, MPI_INT, srcID, MESSTAG, ↩
                  MPI_COMM_WORLD, &status);
              int recvd;
              MPI_Get_count (&status, MPI_INT, &recvd);
              inCommBuff += recvd;
            }
          phase--;
          srcID = rank ^ (1 << (phase - 1));
        }

      // then send towards the gathering root
      int destID = rank ^ (1 << (sendPhase - 1));
      printf ("DEST of #%i : %i\n", rank, destID);

      MPI_Send (commBuff, inCommBuff, MPI_INT, destID, MESSTAG, ↩
          MPI_COMM_WORLD);
    }

  MPI_Finalize ();
  return 0;
}
```

8. Write a program that performs scattering as efficiently as possible, using point-to-point communications, i.e. the equivalent of `MPI_Scatter`. What is the *volume* of data collectively communicated, if each process is to receive $K$ number of items? Express this number as a function of $K$ and the number of processes $N$.

   **Answer**

   The following program requires as a command-line parameter, the rank of the process/node that will scatter the data. The algorithm mirrors the broadcasting algorithm used in Exercise 5.5, as far as the tree of nodes

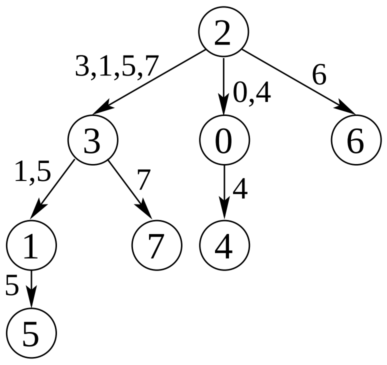


Figure 5.1: An example of a scattering operation initiated from node 2. The edges are labeled with the parts sent in a single operation.

that is implicitly formed is concerned.

The major difference is that the data to be scattered are rearranged in the root node, as the data sent-down a branch of the scatter "tree" are not related to consecutive ranking nodes. Figure 5.1 illustrates a scattering operation initiated from node 2 in an eight-node communicator.

The recursive `findOrder()` function of lines 31-43, calculates the order with which the parts need to be rearranged, so that parts collectively sent down a tree branch, occupy a contiguous buffer space.

For the example shown in Figure 5.1, the parts residing in the `scatteredData` array are rearranged accordingly to form the sequence : 2 3 1 5 7 0 4 6.

```
1  #include<mpi.h>
2  #include<string.h>
3  #include<stdio.h>
4  #include<stdlib.h>
5  #include<iostream>
6  #include<vector>
7  #include<math.h>
8
9  using namespace std;
10
11 #define MESSTAG 0
12 #define MESSSIZE 10
13
14 //****************************************
15 // Returns the number of set bits in its argument
16 int bitCount (int i)
17 {
18   int count = 0;
19   while (i != 0)
20     {
21       int j = i;
22       i >>= 1;
23       if (i != j)
24         count++;
25     }
26   return count;
27 }
28
29 //****************************************
30 // used to calculate in what order the data to be scattered ↩
       should be arranged
31 void findOrder (int ID, int phase, int N, vector < int >&v)
32 {
  if (ID >= N)
    return;
  v.push_back (ID);
  int nextID = ID ^ (1 << phase);
  while (nextID < N)
    {
      findOrder (nextID, phase + 1, N, v);
      phase++;
      nextID = ID ^ (1 << phase);
    }
}

//*****************************************

int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, num, i;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Status status;
  int root = atoi (argv[1]);
  int *data;  // data to be scattered

  int totalPhases = ceil (log2 (num));     // total communication ↩
      phases

  int sendPhase = bitCount (rank ^ root); // phase when a node ↩
      forwards its part of the data

  // allocate the temp. buffer for holding the data to be ↩
      communicated
  int buffSpace = MESSSIZE << (totalPhases - sendPhase);
  if (rank == root)
    buffSpace = num * MESSSIZE;

  int *scatteredData = new int[buffSpace];

  if (rank == root)
    {
      data = new int[MESSSIZE * num];
      for (int i = 0; i < buffSpace; i++)
        data[i] = i;   // something simple to enable verification↩
             of correct receipt

      vector < int >reOrder;
      findOrder (root, 0, num, reOrder);

      // re-arrange the data to be scattered
      int pos = 0;
      for (int i = 0; i < num; i++)
        {
          int destPartIdx = reOrder[i];

          for (int j = 0; j < MESSSIZE; j++)
            scatteredData[pos++] = data[j + destPartIdx * ↩
                MESSSIZE];
        }

      // keep the data that destined for the root of the scatter ↩
          tree
      memcpy (data, scatteredData, MESSSIZE * sizeof (int));

      int phase = 0;
      int buffPos = MESSSIZE;   // skip the part assigned to the ↩
          root
      while (phase <= totalPhases)
        {
          int toSend = MESSSIZE << (totalPhases - phase - 1);
          int destID = rank ^ (1 << phase);

          if (destID < num)
            {
              printf ("#%i sends to %i (%i) at %i\n", rank, ↩
                  destID, toSend, buffPos);
              MPI_Send (scatteredData + buffPos, toSend, MPI_INT,↩
                   destID, MESSTAG, MPI_COMM_WORLD);
              buffPos += toSend;
            }
          phase++;
        }
    }
  else
    {
      data = new int[MESSSIZE];
      // calculate the ID of the node that will send the data to ↩
          be scattered
      int srcID = rank ^ (1 << (sendPhase - 1));
      int toRecv = MESSSIZE << (totalPhases - sendPhase);

      printf ("#%i getting from %i (%i)\n", rank, srcID, toRecv);

      MPI_Recv (scatteredData, toRecv, MPI_INT, srcID, MESSTAG, ↩
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      memcpy (data, scatteredData, MESSSIZE * sizeof (int));    ↩
          // keep the data belonging to the node

      // scatter to other nodes now
      int phase = sendPhase + 1;
      int destID = rank ^ (1 << (phase - 1));
      int buffPos = MESSSIZE;   // skip the part assigned to the ↩
          root
      int toSend = toRecv >> 1;
      while (phase <= totalPhases)
        {
          int destID = rank ^ (1 << phase - 1);
          if (destID < num)
            {
              printf ("#%i sends to %i (%i)\n", rank, destID, ↩
                  toSend);
              MPI_Send (scatteredData + buffPos, toSend, MPI_INT,↩
                   destID, MESSTAG, MPI_COMM_WORLD);
              buffPos += toSend;
            }
          phase++;
          toSend >>= 1;
        }
    }
  // verify scattering
  printf ("%i finished : ", rank);
  for (int i = 0; i < MESSSIZE; i++)
    printf ("%i ", data[i]);
  printf ("\n");

  MPI_Finalize ();
  delete [] data;
  delete [] scatteredData;
  return 0;
}
```

The scattering is performed over $\lceil lg(N) \rceil$ number of steps. In each step, the participating nodes grow exponentially, while the communicated parts shrink exponentially (by a factor of 2 in both cases).

If we number the communication phases $i = 0, \ldots, \lceil lg(N) \rceil - 1$, then we have:

- Phase 0: participating nodes $2^0$, data communicated $\frac{K \cdot N}{2}$
- Phase 1: participating nodes $2^1$, data communicated per node $\frac{K \cdot N}{2^2}$
- Phase 2: participating nodes $2^2$, data communicated per node $\frac{K \cdot N}{2^3}$
- ...

Therefore, the overall data volume is:

$$\sum_{i=0}^{\lceil lg(N)\rceil-1} 2^i \frac{K\cdot N}{2^{i+1}} = \sum_{i=0}^{\lceil lg(N)\rceil-1} \frac{K\cdot N}{2} = \frac{K\cdot N}{2}\lceil lg(N)\rceil \tag{5.3}$$

9. The amount of data exchanged during every step of the butterfly pattern in Figure 5.9, double in relation to the previous step. If initially every process had data of size $K$ bytes to exchange, what is the total time required for the operation to complete, if we assume that each message exchange takes time $t_s + l\cdot V$, where $t_s$ is the link's start-up latency, $V$ is the volume of data to be sent and $l$ is the inverse of the communication speed.

   **Answer**

   The butterfly operation is completed in $lg(N)$ steps, with each step $i = 0,\ldots,lg(N)-1$, requiring the exchange of $2^iK$ data moving in opposite directions. If we assume that there is no delay between the steps, and that communications are bi-directional, then the overall time would be equal to:

$$\begin{aligned}\sum_{i=0}^{lg(N)-1} t_s + l\cdot 2^iK = \\ lg(N)t_s + l\cdot K\sum_{i=0}^{lg(N)-1} 2^i = \\ lg(N)t_s + l\cdot K(2^{lg(N)}-1) = \\ lg(N)t_s + l\cdot K(N-1)\end{aligned} \tag{5.4}$$

10. An alternative parallel bucket sort algorithm, would have the root process of a $N$-process run, scan the input data and split them into $N$ buckets before scattering the buckets to the corresponding processes. Implement this alternative design and compare its performance with the version presented in Section 5.11.5.

    **Answer**

```
#include<mpi.h>
#include<stdlib.h>
#include<math.h>
#include<iostream>

using namespace std;

const int MIN = 0;
const int MAX = 10000;
//*****************************************
int comp (const void *a, const void *b)
{
  return *(reinterpret_cast < const int *>(a)) -*(↩
      reinterpret_cast < const int *>(b));
}

//*****************************************
void initData (int min, int max, int *d, int M)
{
  srand (time (0));
  for (int i = 0; i < M; i++)
    d[i] = (rand () % (max - min)) + min;
}

//****************************************
int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, N;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &N);
  MPI_Status status;

  if (argc == 1)
    {
      if (rank == 0)
        cerr << "Usage " << argv[0] << " number_of_items\n";
      exit (1);
    }

  int M = atoi (argv[1]);
  int maxItemsPerBucket = M;
  int deliveredItems;
  int bucketRange = ceil (1.0 * (MAX - MIN) / N);
  int *data = new int[M];

  if (rank == 0)
    {
      initData (MIN, MAX, data, M);
      int *buckets = new int[N * maxItemsPerBucket];
      int *bucketOffset = new int[N]; // where do buckets begin?
      int *inBucket = new int[N];     // how many items in each ↩
          one?


      // initialize bucket counters and offsets
      for (int i = 0; i < N; i++)
        {
          inBucket[i] = 0;
          bucketOffset[i] = i * maxItemsPerBucket;
        }

      // split into buckets
      for (int i = 0; i < M; i++)
        {
          int idx = (data[i] - MIN) / bucketRange;
          int off = bucketOffset[idx] + inBucket[idx];
          buckets[off] = data[i];
          inBucket[idx]++;
        }

      MPI_Scatter (inBucket, 1, MPI_INT, &deliveredItems, 1, ↩
          MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Scatterv (buckets, inBucket, bucketOffset, MPI_INT, ↩
          data, maxItemsPerBucket, MPI_INT, 0, MPI_COMM_WORLD);
      qsort (data, deliveredItems, sizeof (int), comp);

      // calculate the offsets of the buckets again, so that they↩
           end-up being contiguous after the gathering
      for (int j = 1; j < N; j++)
        bucketOffset[j] = bucketOffset[j - 1] + inBucket[j - 1];

      MPI_Gatherv (data, deliveredItems, MPI_INT, data, inBucket,↩
           bucketOffset, MPI_INT, 0, MPI_COMM_WORLD);

      // print-out for verification purposes
      for (int i = 0; i < M; i++)
        cout << data[i] << " ";
      cout << endl;

      // release memory
      delete[] buckets;
      delete[] inBucket;
      delete [] bucketOffset;
    }
  else
    {
      // get the size of the data to be sorted locally first
      MPI_Scatter (NULL, 1, MPI_INT, &deliveredItems, 1, MPI_INT,↩
           0, MPI_COMM_WORLD);
      MPI_Scatterv (NULL, NULL, NULL, MPI_INT, data, ↩
          maxItemsPerBucket, MPI_INT, 0, MPI_COMM_WORLD);
      qsort (data, deliveredItems, sizeof (int), comp);
      MPI_Gatherv (data, deliveredItems, MPI_INT, NULL, NULL, ↩
          NULL, MPI_INT, 0, MPI_COMM_WORLD);
    }

  MPI_Finalize ();
  delete [] data;
  return 0;
}
```

11. Write a function that could be used for providing multicasting capabilities to a program, i.e. to be able to send a message to a subset of the processes of a communicator. Use an appropriate collective operation for the task.

**Answer**

```
// Multicasts N elements of type t from address data to the list
// of processes included in the ranks array
// The first element in ranks is assumed to be the root for
// MPI_Bcast as it automatically get rank 0 in the new commun.
void multicast (void *data, int N, MPI_Datatype t, int *ranks, ↩
     int rN)
{
  MPI_Group g, all;
  MPI_Comm mc;

  MPI_Comm_group (MPI_COMM_WORLD, &all);
  MPI_Group_incl (all, rN, ranks, &g);
  int localRank;
  MPI_Group_rank (g, &localRank);

  MPI_Comm_create (MPI_COMM_WORLD, g, &mc);
  if (localRank != MPI_UNDEFINED)
    {
      MPI_Bcast (data, N, t, 0, mc);
      MPI_Comm_free (&mc);
    }

  MPI_Group_free (&g);
  MPI_Group_free (&all);
}
```

12. Write the equivalent of the ping-pong program using RMA functions, and measure the communication speed achieved versus the size of the message used. Compare your results with the data rates accomplished with point-to-point communications.

**Answer**

The following program prints out the average communication time over `REP` tests, for messages ranging in size from 0 to `MAX_MESG` bytes, in increments of 1000.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

using namespace std;

```

```
#define MAX_MESG 1000000

const int REP = 10;

int main (int argc, char *argv[])
{
  int size, rank;
  int namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int mesg_size;
  int tag;
  unsigned char *buffer;
  double start_time, end_time;
  MPI_Status status;
  MPI_Win w;
  MPI_Group otherProc, all;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name (processor_name, &namelen);

  printf ("Process %d of %d on %s\n", rank, size, processor_name)↩
      ;
  if (size != 2)
    {
      printf ("Requires exactly 2 processes to run\n");
      exit (1);
    }

  buffer = new unsigned char[MAX_MESG];
  memset (buffer, rank, MAX_MESG);

  MPI_Comm_group (MPI_COMM_WORLD, &all);
  MPI_Group_excl (all, 1, &rank, &otherProc);
  MPI_Win_create (buffer, MAX_MESG, 1, MPI_INFO_NULL, ↩
      MPI_COMM_WORLD, &w);

  if (rank == 0)
    {
      for (int mesg_size = 0; mesg_size <= MAX_MESG; mesg_size +=↩
          1000)
        {
          start_time = MPI_Wtime ();
          for (int i = 0; i < REP; i++)
            {
              // send
              MPI_Win_start (otherProc, 0, w);
              MPI_Put (buffer, mesg_size, MPI_UNSIGNED_CHAR, 1, ↩
                  0, mesg_size, MPI_CHAR, w);
              MPI_Win_complete (w);

              // wait for response
              MPI_Win_post (otherProc, 0, w);
              MPI_Win_wait (w);
            }

          end_time = MPI_Wtime ();
          printf ("%i %lf\n", mesg_size, (end_time - start_time) ↩
              / 2 / REP);
        }
    }
  else
    {
      // other process mirrors what rank 0 is doing
      for (int mesg_size = 0; mesg_size <= MAX_MESG; mesg_size +=↩
          1000)
        for (int i = 0; i < REP; i++)
          {
            MPI_Win_post (otherProc, 0, w);
            MPI_Win_wait (w);

            MPI_Win_start (otherProc, 0, w);
            MPI_Put (buffer, mesg_size, MPI_UNSIGNED_CHAR, 0, 0, ↩
                mesg_size, MPI_CHAR, w);
            MPI_Win_complete (w);
          }
    }

  delete [] buffer;
  MPI_Win_free (&w);
  MPI_Group_free (&all);
  MPI_Group_free (&otherProc);
  MPI_Finalize ();
  return 0;
}
```

13. Modify the program of Listing 5.7, so that the partitioning of the range depends on the relative speed of the participating nodes. One easy approach is to make the ranges proportional to the CPU operating frequency, or the calculated bogomips. Both numbers are available in the `/proc/cpuinfo` pseudo-file. The master can collect the numbers and reply back to the worker nodes with the calculated ranges. If we represent as $m_i$ the $i$-node's bogomips, the percent of the range $\alpha_i$ that should be assigned to node $i$ can be calculated as $\alpha_i = \frac{m_i}{\sum_{\forall k} m_k}$

    **Answer**

    The master node has to gather the bogomips of the execution platform machines, and calculate the appropriate ranges before sending them back to the nodes. The bogomips can be read from the `/proc/cpuinfo` pseudo-file using a pipe as shown in line 23. The bogomips numbers are gathered by the master node (line 31) before calculating the parts $\alpha_i$ (lines 35-41, identified as `part[]` in the code) and the ranges that should be assigned to each node (lines 43-51). The remaining code is identical to Listing 5.7.

```
1   #include <mpi.h>
2   #include<stdio.h>
3   #include<stdlib.h>
4   #include<string.h>
5
6   #define RANGEMIN 0
7   #define RANGEMAX 1000
8   #define MSGTAG 0
9
10
11  int main (int argc, char **argv)
12  {
13    int rank, num, i;
14    int range[2];
15    MPI_Init (&argc, &argv);
16    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
17    MPI_Comm_size (MPI_COMM_WORLD, &num);
18    MPI_Status status;
19
20    float bogoMIPS;
21    float *allBogo = NULL;
22
23    FILE *f = popen ("/bin/cat /proc/cpuinfo  | /bin/grep bogo | /↩
          usr/bin/head -n 1  | /usr/bin/gawk -F ':' '{print $2}'", "↩
          r");
24    fscanf (f, "%f", &bogoMIPS);
25    pclose (f);
26
27    // array for gathering is only need in rank 0
28    if (rank == 0)
29      allBogo = new float[num];
30
31    MPI_Gather (&bogoMIPS, 1, MPI_FLOAT, allBogo, 1, MPI_FLOAT, 0, ↩
          MPI_COMM_WORLD);
32
```

```
  if (rank == 0)
    {
      float sumBogo = allBogo[0];
      for (i = 1; i < num; i++)
        sumBogo += allBogo[i];

      float part[num];
      for (i = 0; i < num; i++)
        part[i] = allBogo[i] / sumBogo;

      int rng[2 * num];
      int width = RANGEMAX - RANGEMIN;
      rng[0] = RANGEMIN;          // left limit
      rng[1] = rng[0] + width * part[0] - 1;     // right limit
      for (i = 1; i < num; i++)
        {
          rng[i * 2] = rng[i * 2 - 1] + 1;
          rng[i * 2 + 1] = (i == num - 1) ? RANGEMAX : rng[i * 2]↩
               + width * part[i] - 1;
        }

      MPI_Request rq[num - 1];
      for (i = 1; i < num; i++)
        MPI_Isend (rng + i * 2, 2, MPI_INT, i, MSGTAG, ↩
            MPI_COMM_WORLD, &(rq[i - 1]));

      for (i = 1; i < num; i++)
        MPI_Wait (&(rq[i - 1]), &status);

      range[0] = rng[0];          // master's limits
      range[1] = rng[1];

      delete [] allBogo;
    }
  else
    {
      MPI_Request rq;
      MPI_Irecv (range, 2, MPI_INT, 0, MSGTAG, MPI_COMM_WORLD, &↩
          rq);
      MPI_Wait (&rq, &status);
    }

  printf ("Node %i's range : ( %i, %i )\n", rank, range[0], range↩
      [1]);

  MPI_Finalize ();
  return 0;
}
```

14. The butterfly communication scheme that is outlined in Section 5.11.4, is only one of the possible strategies for an all-to-all, or all-reduce data exchange. A different approach would be mandated if the underlying communication infrastructure did not provide the required links, making the procedure inefficient. An example of such an architecture is the ring, where each node is directly connected to just two others.

    (a) Write an MPI program that would implement an efficient all-to-all exchange of data on a ring of machines.

    (b) How many steps would be required in comparison to a butterfly scheme, if the number of nodes/processes were $N = 2^k$?

    (c) If we assume that time taken to send $V$ bytes over a communication link is given by $l \cdot V$, where $l$ is the (inverse of the) link speed in $sec/byte$, how does your algorithm compare against the butterfly scheme in terms of overall communication time?

    **Answer**

(a) The following program contains an `allToAll` function that is equivalent to `MPI_Alltoall`. Assuming that $N$ processes are organized in a ring, each process performs $N-1$ rounds/steps of forwarding its data to the next process (identified by `destID`) and receiving data from the previous process (identified by `srcID`). The data are rearranged so that the first block of `NperNode` items received, should reside with the process. The remaining `N - round - 1` blocks are sent to the next process and so on. The data rearrangement is performed before commencing any communications in lines 24-26. The target data layout depends on the rank of the process.

The MPI datatype of the communicated data is needed so that heterogeneous platforms can be supported, while the native type size (parameter `typeSize`) is needed so that a properly-sized data buffer is allocated for handling the communication traffic.

The communication pattern for the `allToAllGather` function is made up from the same number of steps, but with a reverse flow of communications and each communication carries `NperNode` elements.

```
1  //All-to-all scattering on a ring
2  // Number of processes can be arbitrary
3  #include<mpi.h>
4  #include<string.h>
5  #include<unistd.h>
6  #include<math.h>
7  #include<iostream>
8
9  const int K = 10;
10 const int ALLGATHERTAG = 0;
11 const int ALLSCATTERTAG = 0;
12
13 using namespace std;
14
15 //------------------------------------------------------------
16 // all-to-all scattering. outBuffer/inBuffer are supposed to have N * NperNode elements/space
17 void allToAll (void *outBuffer, int NperNode, int typeSize, MPI_Datatype t, void *inBuffer)
18 {
19   int rank, N;
20   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
21   MPI_Comm_size (MPI_COMM_WORLD, &N);
22
23   // local-destined part of data copied first
24   memcpy (inBuffer + NperNode * typeSize * rank, outBuffer + NperNode * typeSize * rank, NperNode * typeSize);
25
26   // rearrange local data before starting to sent
27   unsigned char *tempBuff = new unsigned char[(N - 1) * NperNode * typeSize];
28   memcpy (tempBuff, outBuffer + NperNode * typeSize * (rank + 1), (N - rank - 1) * NperNode * typeSize);
29   memcpy (tempBuff + (N - rank - 1) * NperNode * typeSize, outBuffer, rank * NperNode * typeSize);
30
31   int round = 1;
32   int toSend = (N - 1) * NperNode;
33   int srcID = (rank == 0) ? N - 1 : rank - 1;
34   int destID = (rank + 1) % N;
35   void *ptr1 = tempBuff, *ptr2 = outBuffer, *aux;
36   while (round < N)
37     {
38       MPI_Send (ptr1, toSend, t, destID, ALLSCATTERTAG, MPI_COMM_WORLD);
39       MPI_Recv (ptr2, toSend, t, srcID, ALLSCATTERTAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
40
```

```
      // keep local-destined part and sent the rest
      int offset = ((rank - round + N) % N) * NperNode * typeSize;
      memcpy (inBuffer + offset, ptr2, NperNode * typeSize);

      // switch buffer pointers
      aux = ptr1;
      ptr1 = ptr2;
      ptr2 = aux;
      ptr1 += NperNode * typeSize;      // shift beginning of buffers to send/receive
      ptr2 += NperNode * typeSize;
      toSend -= NperNode;
      round++;
    }
  delete [] tempBuff;
}

//------------------------------------------------------
// all-to-all gathering. outBuffer is supposed to have
// NperNode elements, while inBuffer has enough space
// for N * NperNode
void allToAllGather (void *outBuffer, int NperNode, int typeSize, MPI_Datatype t, void *inBuffer)
{
  int rank, N;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &N);

  // local-destined part of data copied first
  memcpy (inBuffer + NperNode * typeSize * rank, outBuffer, NperNode * typeSize);


  int round = 1;
  // reverse ring traversal than allToAll
  int srcID = (rank + 1) % N;
  int destID = (rank == 0) ? N - 1 : rank - 1;
  void *ptr1 = inBuffer + NperNode * typeSize * rank;
  void *ptr2 = (rank == N - 1) ? inBuffer : ptr1 + NperNode * typeSize;
  void *aux;
  while (round < N)
    {
      MPI_Send (ptr1, NperNode, t, destID, ALLGATHERTAG, MPI_COMM_WORLD);
      MPI_Recv (ptr2, NperNode, t, srcID, ALLGATHERTAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      // advance buffer pointers
      aux = ptr2;
      ptr1 = ptr2;
      ptr2 = ((rank + round + 1) % N == 0) ? inBuffer : aux + NperNode * typeSize;
      round++;
    }
}

//****************************************
int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, N;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &N);
  MPI_Status status;


  double *localPart = new double[K * N];
  double *allParts = new double[K * N];

  if (rank == 0)
    cout << "Test for alltoAll\n";
  for (int i = 0; i < K * N; i++)
    localPart[i] = pow (10, rank) + i;

  allToAll (localPart, K, sizeof (double), MPI_DOUBLE, ↩
      allParts);

  sleep (rank);
  cout << rank << " : ";
  for (int i = 0; i < K * N; i++)
    cout << allParts[i] << " ";
  cout << endl;

  MPI_Barrier (MPI_COMM_WORLD);

  if (rank == 0)
    cout << "Test for alltoAllGather\n";
  for (int i = 0; i < K; i++)
    localPart[i] = rank;

  allToAllGather (localPart, K, sizeof (double), MPI_DOUBLE,↩
      allParts);

  sleep (rank);
  cout << rank << " : ";
  for (int i = 0; i < K * N; i++)
    cout << allParts[i] << " ";
  cout << endl;


  MPI_Finalize ();
  delete [] localPart;
  delete [] allParts;
  return 0;
}
```

(b) The number of steps/rounds required is $N-1$ which is obviously much higher than the $lgN$ of the butterfly scheme. However, the data transmitted in every step by the `allToAllGather` function is constant, while in the butterfly algorithm they grow exponentially in size by a factor of 2.

(c) Assuming that the blocks destined for each process have a size of $K$ bytes, and that the communications during a round take place simultaneously, then the total communication costs are:

$$t_{comm}^{(ring)} = \sum_{i=0}^{N-2} l \cdot K = (N-1) \cdot l \cdot K \tag{5.5}$$

$$t_{comm}^{(butterfly)} = \sum_{i=0}^{lgN-1} l \cdot K \cdot 2^i = l \cdot K(2^{lgN} - 1) = (N-1) \cdot l \cdot K \tag{5.6}$$

The $t_{comm}^{(butterfly)} = t_{comm}^{(ring)}$ result may seem surprising, but there is an important factor that has been intentionally ignored in this setting : communication latency. The start-up overhead can be significant. The butterfly pattern can offer substantially lower costs if this is factored-in as well.

15. The case study on Diffusion Limited Aggregation in Section 5.20, decomposes the problem's data on a particle-wise basis. Explore the alternative of partitioning the 2D grid and assigning it to the processes involved.

What are the benefits and drawbacks of this approach? Is the communication pattern involved different in this case? Does this approach produce similar results to the sequential program and alternative parallel programs?

**Answer**

The benefit of a row-wise geometric decomposition is a more regular communication pattern that is also localized, instead of global. Changes to the crystal formation do not have to be broadcasted to all the nodes. The drawbacks are (a) a skewed load distribution, as the nodes handling the crystal region have in general fewer particles assigned to them, and (b) a potentially much higher communication cost that grows linearly with the number of columns used. Results should be identical to the sequential program, as long as the sequential program also grows the crystal at the end of each time step.

In the following listing (which is part of a project including the `dla_core.cpp` file shown later), the grid is initialized in node 0 (line 74) and scattered in a row-wise fashion to the remaining nodes (line 79).

The grid local to each node is made up of two additional columns (to simplify boundary checking) and two additional rows that contain data handled by the neighboring processes. The two top and bottom rows of each local grid are exchanged with the previous and next process respectively (the first and last processes are obviously an exception). The particles that end-up in the top/bottom rows "migrate" to the previous/next process after the message exchanges taking place in line 92-93 and 99-100. The loops of lines 113-115 and 120-122 collect those migrating particles. The `bounceUp` and `bounceDown` variables control whether this migration is possible : It is obviously impossible for the very first and very last rows in the overall grid, because that would mean that the particles are leaving completely.

The final state of the grid is gathered by process 0 (line 134) and output to the console in the form of a PBM image (lines 138-149). A schematic of the problem decomposition is shown in Figure 5.2.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5  #include <math.h>
6  #include <iostream>
7  #include "dla_core.h"
8  #include <mpi.h>
9
10 using namespace std;
11
12 #define PIC_SIZE 100
13 #define PARTICLES 500
14 #define MAX_ITER 10000
15
16 /*--------------------------------------------------------*/
17 int main (int argc, char **argv)
18 {
19   int cols, rows, iter, particles, x, y;
20   int *pic, *pic2, *tmp;
21   int *nextNodeRows, *prevNodeRows;
22   int rank, num, i;
23   MPI_Init (&argc, &argv);
24   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

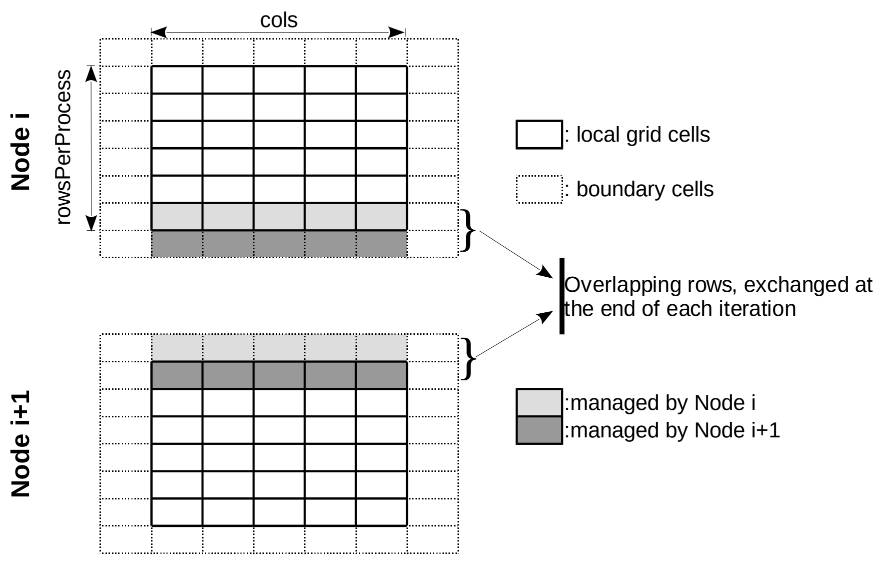


Figure 5.2: Geometric problem decomposition illustration for the solution of Exercise 15.

```
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Request req;

  if (argc < 2)                  // use default values if user ↩
      does not specify anything
    {
      cols = PIC_SIZE;
      rows = PIC_SIZE;
      iter = MAX_ITER;
      particles = PARTICLES;
    }
  else
    {
      cols = atoi (argv[1]);
      rows = atoi (argv[2]);
      particles = atoi (argv[3]);
      iter = atoi (argv[4]);
    }

  // to make code simpler and avoid special cases, rows is forced↩
       to be a multiple of
  // the number of processes
  rows = (int) ceil (rows * 1.0 / num) * num;
  int rowsPerNode = rows / num;

  // initialize the random number generator
  srand(time(0));

  // grid has two extra rows, one above and one below the actual ↩
      rows
  // The extra rows "belong" to other nodes, so at the end of ↩
      each time step
  // there should be a pair-wise reduction of these rows
  if (rank == 0)
    {
      pic = new int[(cols + 2) * (rows + 2)];
      pic2 = new int[(cols + 2) * (rows + 2)];
    }
  else
    {
      pic = new int[(cols + 2) * (rowsPerNode + 2)];
      pic2 = new int[(cols + 2) * (rowsPerNode + 2)];
    }
  bool bounceUp = false, bounceDown = false;
  if (rank == 0)
    bounceUp = true;
  else if (rank == num - 1)
    bounceDown = true;

  nextNodeRows = new int[2 * (cols + 2)];
  prevNodeRows = new int[2 * (cols + 2)];

  if (rank == 0)
    dla_init (pic2, rows + 2, cols + 2, particles, 1);
  // clean pic buffer
  memset (pic, 0, (cols + 2) * (rowsPerNode + 2) * sizeof (int));

  // boundary rows do not need to be communicated now.
  MPI_Scatter (pic2 + cols + 2, (cols + 2) * rowsPerNode, MPI_INT↩
      , pic + cols + 2, (cols + 2) * rowsPerNode, MPI_INT, 0, ↩
      MPI_COMM_WORLD);

  int nextID = rank + 1, prevID = rank - 1;
  while (--iter >= 0)
    {
      dla_evolve (pic, pic2, rowsPerNode + 2, cols + 2, bounceUp,↩
           bounceDown);
      tmp = pic2;
      pic2 = pic;
      pic = tmp;
      //exchange information with neighboring nodes
      // exchange with "previous"
      if (rank != 0)
        {
          MPI_Isend (pic, 2 * (cols + 2), MPI_INT, prevID, iter, ↩
              MPI_COMM_WORLD, &req);
          MPI_Recv (prevNodeRows, 2 * (cols + 2), MPI_INT, prevID↩
              , iter, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

      // exchange with "next"
      if (rank != num - 1)
        {
          MPI_Isend (pic + rowsPerNode * (cols + 2), 2 * (cols + ↩
              2), MPI_INT, nextID, iter, MPI_COMM_WORLD, &req);
          MPI_Recv (nextNodeRows, 2 * (cols + 2), MPI_INT, nextID↩
              , iter, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

      // reduction is duplicated in the nodes so that they dont ↩
          have to wait
      // what is important in the neighbors'rows, is the negative↩
           cells
      if (rank != 0)
        {
          for (int i = 0; i < cols + 2; i++)
            if (prevNodeRows[i] < 0)
              pic[i] = prevNodeRows[i];
            else
              pic[i] = 0;        // reset any particles now on the↩
                   other node

          for (int i = 0; i < cols + 2; i++)
            if (prevNodeRows[cols + 2 + i] > 0)
              pic[cols + 2 + i] += prevNodeRows[cols + 2 + i];   ↩
                  // add particles coming from the other node
        }

      if (rank != num - 1)
        {
          for (int i = 0; i < cols + 2; i++)
            if (nextNodeRows[i] > 0)
              pic[rowsPerNode * (cols + 2) + i] += nextNodeRows[ ↩
                  i];  // add particles coming from the other ↩
                  node

          for (int i = 0; i < cols + 2; i++)
            if (nextNodeRows[cols + 2 + i] < 0)
              pic[(rowsPerNode + 1) * (cols + 2) + i] = nextNodeRows[cols + 2 + i];
            else
              pic[(rowsPerNode + 1) * (cols + 2) + i] = 0;        // reset any particles now on the other node
        }

    }


  MPI_Gather (pic + cols + 2, rowsPerNode * (cols + 2), MPI_INT, pic + cols + 2, rowsPerNode * (cols + 2), MPI_INT, 0, MPI_COMM_WORLD);
  /* Print to stdout a PBM picture of the simulation space */
  if (rank == 0)
    {
      printf ("P1\n%i %i\n", cols, rows);

      for (y = 1; y <= rows; y++)
        {
          for (x = 1; x <= cols; x++)
            {
              if (pic[y * (cols + 2) + x] < 0)
                printf ("1 ");
              else
                printf ("0 ");
            }
          printf ("\n");
        }
    }

  MPI_Finalize ();
  delete[] pic;
  delete[] pic2;
  delete[] nextNodeRows;
  delete[] prevNodeRows;
  return 0;
}
```

The `dla_core.cpp` file that supplements the code above is:

```
#include <stdlib.h>
#include "dla_core.h"

/*---------------------------------------------------
 * Checks the presence of a structure in neighboring cells
 * pic points to 2D array holding data, arranged in cols
 * columns. Because the array is two columns and two rows
 * wider than necessary, there is no need to check for
 * boundary values of x and y.
 */
int check_proxim (int *pic, int cols, int x, int y)
{
  int *row0, *row1, *row2;
  row0 = pic + (y - 1) * cols + x - 1;
  row1 = row0 + cols;
  row2 = row1 + cols;
  if (*row0 < 0 || *(row0 + 1) < 0 || *(row0 + 2) < 0 || *row1 < 0 || *(row1 + 1) < 0 || *(row1 + 2) < 0 || *row2 < 0 || *(row2 + 1) < 0 || *(row2 + 2) < 0)
    return (-1);
  else
    return (1);
}

/*---------------------------------------------------*/
/* Returns -1,0 and 1 with equal probability */
inline int three_way ()
{
  return (random () % 3) - 1;
```

```c
}

/*--------------------------------------------------------*/
/* Initializes the 2D array for the simulation */
void dla_init (int *pic, int rows, int cols, int particles, int init_seed)
{
  int i, j, x, y;
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
      pic[i * cols + j] = 0;


  for (i = 0; i < particles; i++)       /* generate initial particle placement */
    {
      x = random () % (cols - 2) + 1;    // counting starts from 1
      y = random () % (rows - 2) + 1;
      if ((y == rows / 2 + 1) && (x == cols / 2 + 1))    // repeat if true
        i--;
      else
        pic[y * cols + x]++;
    }

  if (init_seed)
    pic[(rows / 2) * cols + (cols / 2)] = -1;   /* place initial seed */

}

/*--------------------------------------------------------
 * Single step evolution of the simulation.
 *
 * The cell values represent:
 *   0: empty space
 *   >0 : multiple particles
 *   <0 : crystal
 *
 * Returns the address of the structure holding the last update */
int *dla_evolve (int *pic, int *pic2, int rows, int cols, bool bounceUp, bool bounceDown)
{
  int x, y, k;

  // prepare array to hold new state
  for (y = 1; y < rows - 1; y++)
    for (x = 1; x < cols - 1; x++)
      pic2[y * cols + x] = pic[y * cols + x] > 0 ? 0 : pic[y * cols + x];

  for (y = 1; y < rows - 1; y++)
    for (x = 1; x < cols - 1; x++)
      for (k = 0; k < pic[y * cols + x]; k++)
        {
          int new_x = x + three_way ();
          if (new_x == 0 && bounceUp)
            new_x = 1;
          else if (new_x == cols - 1 && bounceDown)
            new_x = cols - 2;

          int new_y = y + three_way ();
          if (new_y == 0)
            new_y = 1;
          else if (new_y == rows - 1)
            new_y = rows - 2;

          if (pic2[new_y * cols + new_x] > 0)    // steps into empty space
            pic2[new_y * cols + new_x]++;
          else if (pic2[new_y * cols + new_x] == 0)      // steps into unchecked space
            {
                pic2[new_y * cols + new_x] = check_proxim (pic2, cols, new_x, new_y);
              }
          }
  return pic2;
}
```

16. Implement a 3D-space Diffusion Limited Aggregation simulation, by extending the solutions provided in Section 5.20.

    **Answer**

    The project reported in Section 5.20 consists of two source code files: `dla_core.c` and `dla_mpi.c`. The transition to 3D requires significant changes only to the functions in `dla_core.c`, which handle the particles as represented by the `PartStr` structure (which in turn gains a `z` coordinate).

    The complete code (with a minor conversion to C++) for `dla_core.cpp` is shown below, the major differences being limited to:

    (a) Lines 24-33 of the `check_proxim` function, that now has to check 27 possible neighboring positions, scattered over 3 planes along the z-axis.

    (b) The calculation of the indices into the grid array (as seen in lines 69, 109 and 142), that need to take into consideration the third dimension.

```
1  /* Core library for simulating 3D DLA phenomena */
2  #include <stdlib.h>
3  #include <string.h>
4  #include "dla_core.h"
5
6  /*--------------------------------------------------
7   * Checks the presence of a structure in neighboring cells
8   * pic points to 3D array holding data, arranged in depth number
9   * of colsXcolumns planes. Because the array is "surrounded" by
10  * an empty cube of cells, there is no need to check for
11  * boundary values of x, y and z.
12  */
13 int check_proxim (int *pic, int rows, int cols, int x, int y, int z)
14 {
15   int *plane0, *plane1, *plane2;
16   plane0 = pic + (z - 1) * rows * cols;
17   plane1 = pic + z * rows * cols;
18   plane2 = pic + (z + 1) * rows * cols;
19
20   int row0, row1, row2;
21   row0 = (y - 1) * cols + x - 1;
22   row1 = row0 + cols;
23   row2 = row1 + cols;
24   if (plane0[row0] < 0 || plane0[row0 + 1] < 0 || plane0[row0 + 2] < 0 ||
25       plane0[row1] < 0 || plane0[row1 + 1] < 0 || plane0[row1 + 2] < 0 ||
26       plane0[row2] < 0 || plane0[row2 + 1] < 0 || plane0[row2 + 2] < 0 ||
27       plane1[row0] < 0 || plane1[row0 + 1] < 0 || plane1[row0 + 2] < 0 ||
28       plane1[row1] < 0 || plane1[row1 + 1] < 0 || plane1[row1 + 2] < 0 ||
29       plane1[row2] < 0 || plane1[row2 + 1] < 0 || plane1[row2 + 2] < 0 ||
30       plane2[row0] < 0 || plane2[row0 + 1] < 0 || plane2[row0 + 2] < 0 ||
31       plane2[row1] < 0 || plane2[row1 + 1] < 0 || plane2[row1 + 2] < 0 ||
```

```c
   plane2[row2] < 0 || plane2[row2 + 1] < 0 || plane2[row2 + 2] < 0)
  return (-1);
 else
  return (1);
}

/*--------------------------------------------------*/
/* Returns -1,0 and 1 with equal probability */
inline int three_way ()
{
 return (random () % 3) - 1;
}

/*--------------------------------------------------*/
/* Initializes the 3D array and array of particles for the simulation */
void dla_init_plist (int *pic, int rows, int cols, int depth, PartStr * p, int particles, int init_seed)
{
 int i, x, y, z, idx;
 memset (pic, 0, rows * cols * depth * sizeof (int));


 for (i = 0; i < particles; i++)       /* generate initial particle placement */
  {
   x = random () % (cols - 2) + 1;    // counting starts from 1
   y = random () % (rows - 2) + 1;
   z = random () % (rows - 2) + 1;
   if ((y == rows / 2 + 1) && (x == cols / 2 + 1) && (z == depth / 2 + 1))   // repeat if true
    i--;
   else
    {
     p[i].x = x;
     p[i].y = y;
     p[i].z = z;
    }
  }
 if (init_seed)
  {
   idx = (depth / 2) * rows * cols + (rows / 2) * cols + (cols / 2);
   pic[idx] = -1;             /* place initial seed */
  }

}

/*--------------------------------------------------
 * Single step evolution of the simulation.
 *
 * The cell values represent:
 *  0  : empty space
 *  <0 : crystal  -- for consistency with alternative formulation
 *
 * Particles are held in a separate array.
 */
void dla_evolve_plist (int *pic, int rows, int cols, int depth, PartStr * p, int *particles, PartStr * changes, int *numChanges)
{
 int i;
 *numChanges = 0;

 for (i = 0; i < *particles; i++)
  {
   int new_x = p[i].x + three_way ();
   if (new_x == 0)            // bounce off boundaries
    new_x = 1;
   else if (new_x == cols - 1)
    new_x = cols - 2;

   int new_y = p[i].y + three_way ();
 98       if (new_y == 0)            // bounce off boundaries
 99         new_y = 1;
100       else if (new_y == rows - 1)
101         new_y = rows - 2;
102
103       int new_z = p[i].z + three_way ();
104       if (new_z == 0)            // bounce off boundaries
105         new_z = 1;
106       else if (new_z == depth - 1)
107         new_z = depth - 2;
108
109       int idx = new_z * rows * cols + new_y * cols + new_x;
110       if (pic[idx] == 0)         // steps into empty space
111         {
112           int turnCrystal = check_proxim (pic, rows, cols, new_x, new_y, new_z);
113           if (turnCrystal < 0)
114             {
115               // record crystal change
116               changes[*numChanges].x = new_x;
117               changes[*numChanges].y = new_y;
118               changes[*numChanges].z = new_z;
119               (*numChanges)++;
120
121               // erase particle from list
122               p[i] = p[(*particles) - 1];
123               i--;
124               (*particles)--;
125             }
126           else                   // change position to particle
127             {
128               p[i].x = new_x;
129               p[i].y = new_y;
130               p[i].z = new_z;
131             }
132         }
133     }
134 }
135
136 /*---------------------------------------------------*/
137 void apply_changes (int *pic, int rows, int cols, PartStr * changes, int numChanges)
138 {
139   int i;
140   for (i = 0; i < numChanges; i++)
141     {
142       int idx = changes[i].z * rows * cols + changes[i].y * cols + changes[i].x;
143       pic[idx] = -1;
144     }
145 }
```

On the other hand, the contents of `dla_mpi.cpp` file shown below, require only minor changes. These are limited to:

(a) Lines 206-220, where the derived MPI datatype for `PartStr` is defined.

(b) Lines 247-258 that output the resulting structure as a CSV file. This CSV file can be imported in ParaView (http://www.paraview.org/Wiki/ParaView/Data_formats) for viewing purposes.

```
146 /* DLA 3D MPI program
147  * Particles are evenly distributed among nodes/processes
148  * G. Barlas, 2014
149  */
150
151 #include <stdio.h>
152 #include <stdlib.h>
153 #include <assert.h>
154 #include "dla_core.h"
```

```c
#include <mpi.h>

#define PIC_SIZE 100
#define PARTICLES 1000
#define MAX_ITER 10000

/*-----------------------------------------------------*/
int main (int argc, char **argv)
{
  int cols, rows, depth, iter, particles, x, y;
  int *pic;
  PartStr *p, *changes, *totalChanges;
  int rank, num, i, numChanges, numTotalChanges;
  int *changesPerNode, *buffDispl;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);

  if (argc < 2)                // use default values if user ↩
      does not specify anything
    {
      cols = PIC_SIZE + 2;
      rows = PIC_SIZE + 2;
      depth = PIC_SIZE + 2;
      iter = MAX_ITER;
      particles = PARTICLES;
    }
  else
    {
      cols = atoi (argv[1]) + 2;
      rows = atoi (argv[2]) + 2;
      depth = atoi (argv[3]) + 2;
      particles = atoi (argv[4]);
      iter = atoi (argv[5]);
    }

  // initialize the random number generator
  srand(time(0));

  int particlesPerNode = particles / num;
  if (rank == num - 1)
    particlesPerNode = particles - particlesPerNode * (num - 1); ↩
           // in case particles cannot be split evenly
// printf("%i has %i\n", rank, particlesPerNode);
  pic = (int *) malloc (sizeof (int) * cols * rows * depth);
  p = (PartStr *) malloc (sizeof (PartStr) * particlesPerNode);
  changes = (PartStr *) malloc (sizeof (PartStr) * ↩
      particlesPerNode);
  totalChanges = (PartStr *) malloc (sizeof (PartStr) * ↩
      particlesPerNode);
  changesPerNode = (int *) malloc (sizeof (int) * num);
  buffDispl = (int *) malloc (sizeof (int) * num);
  assert (pic != 0 && p != 0 && changes != 0 && totalChanges != 0↩
       && changesPerNode != 0);

  // MPI user type declaration
  int lengths[3] = { 1, 1, 1 };
  MPI_Datatype types[3] = { MPI_INT, MPI_INT, MPI_INT };
  MPI_Aint add1, add2, add3;
  MPI_Aint displ[3];
  MPI_Datatype Point;

  MPI_Address (p, &add1);
  MPI_Address (&(p[0].y), &add2);
  MPI_Address (&(p[0].z), &add3);
  displ[0] = 0;
  displ[1] = add2 - add1;
  displ[2] = add3 - add1;

  MPI_Type_struct (3, lengths, displ, types, &Point);
  MPI_Type_commit (&Point);

  dla_init_plist (pic, rows, cols, depth, p, particlesPerNode, 1)↩
      ;
  while (--iter)
    {
      dla_evolve_plist (pic, rows, cols, depth, p, &↩
          particlesPerNode, changes, &numChanges);

      //exchange information with other nodes
      MPI_Allgather (&numChanges, 1, MPI_INT, changesPerNode, 1, ↩
          MPI_INT, MPI_COMM_WORLD);
      //calculate offsets
      numTotalChanges = 0;
      for (i = 0; i < num; i++)
        {
          buffDispl[i] = numTotalChanges;
          numTotalChanges += changesPerNode[i];
        }

        if (numTotalChanges > 0)
        {
          MPI_Allgatherv (changes, numChanges, Point, ↩
              totalChanges, changesPerNode, buffDispl, Point, ↩
              MPI_COMM_WORLD);
          apply_changes (pic, rows, cols, totalChanges, ↩
              numTotalChanges);
        }
    }

  if (rank == 0)
    {
      // save data points as a CSV file that can be imported in ↩
          Paraview
      FILE *f = fopen ("dla.csv", "w+t");
      fprintf (f, "x coord, y coord, z coord, scalar\n");
      int idx = 0;
      for (int z = 0; z < depth; z++)
        for (int y = 0; y < rows; y++)
          for (int x = 0; x < cols; x++)
            {
              if (pic[idx] == -1)
                fprintf (f, "%i, %i, %i, 1\n", x, y, z);
              idx++;
            }
      fclose (f);
    }

  MPI_Reduce (&particlesPerNode, &particles, 1, MPI_INT, MPI_SUM,↩
        0, MPI_COMM_WORLD);
  if (rank == 0)
    fprintf (stderr, "Remaining particles %i\n", particles);

  free (pic);
  free (p);
  free (changes);
  free (changesPerNode);
  free (buffDispl);
  MPI_Finalize ();
  return 0;
}
```

17. It is not unusual in NoWs setups, to have machines with different capabilities. Create an MPI program that would have each process read from a file a number of integers that is proportional to its CPU speed as indicated by the operating frequency (its “clock”). Use appropriate *filetypes* and *views* to perform the data distribution in a cyclic-block manner.

    **Answer**

    Assuming that the input file is broken into `BLOCKSIZE` pieces [1], the solution

[1]Actually, it is an implicit requirement that the number of etype elements in the file is a multiple of the filetype which is in our case equal to BLOCKSIZE in length. Otherwise, the “excess” part of the file will not be read by any process.

is assign to each node, a part of a block proportional to its clock speed.

The mathematical formulation is identical to the one used in Exercise 6 and Equations 5.1 and 5.2. Each node $i$ creates a filetype that corresponds to a part of block equal to

$$localSize_i = \lfloor part_i \cdot BLOCKSIZE \rfloor \tag{5.7}$$

in size, that starts at offset

$$localOff_i = \sum_{j=0}^{i-1} \lfloor part_j \cdot BLOCKSIZE \rfloor \tag{5.8}$$

from the beginning of a block.

In order to make sure that the sum of all parts equals `BLOCKSIZE`, the last of $N$ nodes is assigned a part equal to

$$localSize_{N-1} = BLOCKSIZE - \sum_{j=0}^{N-2} \lfloor part_j \cdot BLOCKSIZE \rfloor \tag{5.9}$$

The resulting source code is shown below. Lines 24-31 deal with the local retrieval and global all-to-all distribution of the clock frequencies. The all-to-all scattering operation is required so that the computation of the individual parts can be duplicated in all nodes, without the need to have them communicated by the master node.

Lines 33-57 implements Equations 5.1, 5.2, 5.7 and 5.8. Lines 55, 56, executed solely on the last node of the communicator, deal with the truncations issues depicted by Equation 5.9.

Lines 65 and 66 declare a different local filetype based on the characteristics of each node. This in turn is used to read the data, one filetype at a time, in lines 77-90.

Commented-out lines 67, 81 and 87 represent alternative ways for accomplishing the desired results.

```
1  // Works only if the filesize is a multiple of the calculated ↩
       actualBlock
2  #include <mpi.h>
3  #include<stdio.h>
4  #include<stdlib.h>
5  #include<string.h>
6  #include<unistd.h>
7  #include<vector>
8
9  using namespace std;
10
11 #define BLOCKSIZE 1000
12
13 int main (int argc, char **argv)
14 {
15   int rank, num, i;
16   MPI_Init (&argc, &argv);
17   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
18   MPI_Comm_size (MPI_COMM_WORLD, &num);
19   MPI_Status status;
20
```

```
  float clockMHz;
  float *allClock = NULL;

  FILE *fp = popen ("/bin/cat /proc/cpuinfo  | /bin/grep MHz | /usr/bin/head -n 1  | /usr/bin/gawk -F ':' '{print $2}'", "r");
  fscanf (fp, "%f", &clockMHz);
  pclose (fp);

  allClock = new float[num];

  // all gather so calculations can be replicated across all nodes
  MPI_Allgather (&clockMHz, 1, MPI_FLOAT, allClock, 1, MPI_FLOAT, MPI_COMM_WORLD);

  float sumClocks = allClock[0];
  for (i = 1; i < num; i++)
    sumClocks += allClock[i];

  float part[num];
  for (i = 0; i < num; i++)
    part[i] = allClock[i] / sumClocks;

  int localSize;
  int localOff = 0;
  int actualBlock = 0;
  for (i = 0; i < rank; i++)
    {
      localOff += part[i] * BLOCKSIZE;
      actualBlock += part[i] * BLOCKSIZE;
    }
  localSize = part[rank] * BLOCKSIZE;
  actualBlock += localSize;
  for (i = rank + 1; i < num; i++)
    actualBlock += part[i] * BLOCKSIZE;

  // last node picking up the slack, so that actualBlock == BLOCKSIZE
  if(rank == num-1)
      localSize += ( BLOCKSIZE - actualBlock);
  actualBlock = BLOCKSIZE;

  MPI_File f;
  MPI_File_open (MPI_COMM_WORLD, argv[1], MPI_MODE_RDONLY, MPI_INFO_NULL, &f);

  MPI_Datatype filetype;
  int starts = 0;

  MPI_Type_create_subarray (1, &actualBlock, &localSize, &starts, MPI_ORDER_C, MPI_INT, &filetype);
  MPI_Type_commit (&filetype);
//   MPI_File_set_view (f, 0, MPI_INT, filetype, (char *)"native", MPI_INFO_NULL);
  MPI_File_set_view (f, localOff * sizeof (int), MPI_INT, filetype, (char *) "native", MPI_INFO_NULL);

  vector < int >data;
  int temp[BLOCKSIZE];

  MPI_Offset filesize;
  MPI_File_get_size (f, &filesize);     // get size in bytes
  filesize /= sizeof (int);     // convert size in number of items
  long long int pos = 0;        //localOff;       // initial file position per process
  while (pos < filesize)
    {
      MPI_File_read (f, temp, 1, filetype, &status);
      int cnt;
//       MPI_Get_count (&status, filetype, &cnt);
      // get the number of data read in etype units
      MPI_Get_count (&status, MPI_INT, &cnt);


      pos += actualBlock;
//        for (int i = 0; i < localSize; i++)
      for (int i = 0; i < cnt; i++)
        data.push_back (temp[i]);
    }

  sleep (rank);
  cout << rank << " read " << data.size () << " numbers." << endl;
  for (int i = 0; i < 30; i++)
    cout << data[i] << " ";
  cout << ".... Last one is : " << data[data.size () - 1];
  cout << endl;

  MPI_Finalize ();
  return 0;
}
```

18. The details of the trapezoidal rule for computing a function integral are discussed in Section 3.5.2. Implement an MPI-based version of the trapezoidal rule using: (a) dynamic partitioning and (b) static partitioning. For part (a) you can base your solution on master-worker implementation of Section 5.22.1.

    **Answer**

    (a) The following listing is a derivation of the master-worker implementation of Section 5.22.1. Given the simple nature of the problem and the associated I/O, the resulting code is much simpler than the original. The only essential customizations are: the adaptation of the derived datatype used to communicate the description of a "job", and the elimination of the "job assignment tracking" feature (supported by the `assignedPart` array in Listing 5.38), as in this case the results are scalar, floating point numbers that are accumulated by the master node (line 121).

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <iostream>
4   #include <math.h>
5   #include <mpi.h>
6   using namespace std;
7
8   //*************************************************************
9   // Communication tags
10  #define NULLRESULTTAG 0
11  #define RESULTTAG     1
12  #define WORKITEMTAG   2
13  #define ENDTAG        3
14
15  //*************************************************************
16  struct WorkItem
17  {
18    double start;
19    double end;
20    int divisions;
21  };
22
23  //-----------------------------------------------
24  double func (double x)
25  {
26    return fabs (sin (x));
27  }
28
29  //-----------------------------------------
30  double integrCalc (double st, double en, int div)
31  {
  //calculate area
  double localRes = 0;
  double step = (en - st) / div;
  double x;
  x = st;
  localRes = func (st) + func (en);
  localRes /= 2;
  for (int i = 1; i < div; i++)
    {
      x += step;
      localRes += func (x);
    }
  localRes *= step;
  return localRes;
}

//**********************************************************
void registerWorkItem (MPI_Datatype * workItemType)
{
  struct WorkItem sample;

  int blklen[3] = { 1, 1, 1 };
  MPI_Aint displ[3], off, base;
  MPI_Datatype types[3] = { MPI_DOUBLE, MPI_DOUBLE, MPI_INT };

  displ[0] = 0;
  MPI_Get_address (&(sample.start), &base);
  MPI_Get_address (&(sample.end), &off);
  displ[1] = off - base;
  MPI_Get_address (&(sample.divisions), &off);
  displ[2] = off - base;

  MPI_Type_create_struct (3, blklen, displ, types, workItemType);
  MPI_Type_commit (workItemType);
}

//**********************************************************
int main (int argc, char *argv[])
{
  int N, rank;
  double start_time, end_time;
  MPI_Status status;
  MPI_Request request;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &N);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  MPI_Datatype workItemType;
  registerWorkItem (&workItemType);

  if (rank == 0)                  // master code
    {
      double LOWERLIMIT = 0;
      double UPPERLIMIT = 10;
      int NDIV = 1000000;
      int NJOBS = 10;

      if (argc > 4)
        {
          LOWERLIMIT = atof (argv[1]);
          UPPERLIMIT = atof (argv[2]);
          NDIV = atoi (argv[3]);
          NJOBS = atoi (argv[4]);
        }

      double total = 0, partialRes;
      int divPerJob = NDIV / NJOBS;
      WorkItem *w = new WorkItem[NJOBS];
      double jobStep = (UPPERLIMIT - LOWERLIMIT) / NJOBS;
      double jobSt = LOWERLIMIT;
      double jobEnd = jobSt + jobStep;
      for (int i = 0; i < NJOBS; i++)
        {
```

```
106          w[i].start = jobSt;
107          w[i].end = jobEnd;
108          w[i].divisions = divPerJob;
109          jobSt = jobEnd;
110          jobEnd += jobStep;
111        }
112
113      // now distribute the work item to the worker nodes
114      for (int i = 0; i < NJOBS; i++)
115        {
116          MPI_Recv (&partialRes, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ↩
                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
117          int workerID = status.MPI_SOURCE;
118          int tag = status.MPI_TAG;
119          MPI_Isend (&(w[i]), 1, workItemType, workerID, ↩
                 WORKITEMTAG, MPI_COMM_WORLD, &request);
120          if (tag == RESULTTAG)
121            total += partialRes;
122        }
123
124      // now send termination messages
125      for (int i = 1; i < N; i++)
126        {
127          MPI_Recv (&partialRes, 1, MPI_DOUBLE, MPI_ANY_SOURCE, ↩
                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
128          int workerID = status.MPI_SOURCE;
129          int tag = status.MPI_TAG;
130
131          if (tag == RESULTTAG)
132            total += partialRes;
133          MPI_Isend (NULL, 0, workItemType, workerID, ENDTAG, ↩
                 MPI_COMM_WORLD, &request);
134        }
135
136      cout << "Result is " << total << endl;
137      delete [] w;
138    }
139  else                          // worker code
140    {
141      MPI_Send (NULL, 0, MPI_DOUBLE, 0, NULLRESULTTAG, ↩
             MPI_COMM_WORLD); // establish communication with ↩
             master
142      while (1)
143        {
144          WorkItem w;
145          MPI_Recv (&w, 1, workItemType, 0, MPI_ANY_TAG, ↩
                 MPI_COMM_WORLD, &status);      // get a new work ↩
                 item
146          int tag = status.MPI_TAG;
147          if (tag == ENDTAG)
148            break;
149
150          double partRes = integrCalc (w.start, w.end, w.↩
                 divisions);
151          MPI_Send (&partRes, 1, MPI_DOUBLE, 0, RESULTTAG, ↩
                 MPI_COMM_WORLD);     // return the results
152        }
153    }
154
155  MPI_Finalize ();
156  return 0;
157 }
```

(b) The static partitioning is a much simpler proposition. Each node can query the command-line parameters and determine (without receiving any information from the "master") the part of the integration range that is dedicate to it (lines 211, 212). Following the partial integration of line 213, with a reduction (line 216) completes the calculation.

```
158 #include <stdio.h>
159 #include <stdlib.h>
160 #include <iostream>
```

```
#include <math.h>
#include <mpi.h>
using namespace std;

//------------------------------------------------------------
double func (double x)
{
  return fabs (sin (x));
}

//------------------------------------------------------
double integrCalc (double st, double en, int div)
{
  //calculate area
  double localRes = 0;
  double step = (en - st) / div;
  double x;
  x = st;
  localRes = func (st) + func (en);
  localRes /= 2;
  for (int i = 1; i < div; i++)
    {
      x += step;
      localRes += func (x);
    }
  localRes *= step;
  return localRes;
}

//*************************************************************
int main (int argc, char *argv[])
{
  int N, rank;
  double LOWERLIMIT = 0;
  double UPPERLIMIT = 10;
  int NDIV = 1000000;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &N);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  if (argc > 3)
    {
      LOWERLIMIT = atof (argv[1]);
      UPPERLIMIT = atof (argv[2]);
      NDIV = atoi (argv[3]);
    }

  int divPerNode = NDIV / N;
  double nodeStep = (UPPERLIMIT - LOWERLIMIT) / N;
  double localSt = LOWERLIMIT + rank * nodeStep;
  double localEnd = localSt + nodeStep;
  double localRes = integrCalc (localSt, localEnd, divPerNode);
  double total;

  MPI_Reduce (&localRes, &total, 1, MPI_DOUBLE, MPI_SUM, 0, ↩
      MPI_COMM_WORLD);
  if (rank == 0)
    cout << "Result is " << total << endl;

  MPI_Finalize ();
  return 0;
}
```

19. Use the master-worker code provided in Section 5.22, as a basis for the creation of a hierarchical master-worker configuration, where nodes are organized in a 3-level (or higher) tree instead of the 2-level tree of the simple setup. The secondary master nodes of the middle tree layer(s) should be responsible for managing the load distribution in their subtrees, while the primary master at the root of the tree should be responsible for

the overall workload distribution.

**Answer**

The calculation of the Mandelbrot fractal is by no means a suitable target application for this kind of setup. So this solution is more of a proof-of-concept, than a practical way to speed-up the particular calculation.

In the listing that follows, nodes form a 3-level hierarchy, with `SECONDARYMASTER` nodes serving in the intermediate layer. The worker nodes calculate the secondary master they are reporting to, with the simple calculation of line 347. The major points can be summarized as follows:

- The master node initially distributes groups of `BATCHSIZE` number of work items (lines 234-242). Subsequently, it sends to each secondary master a single work item for every partial result received (lines 245-255). When all the work items have been distributed, the master sends termination messages for each partial result received (lines 257-266).
- The secondary master nodes maintain a local pool of work items, that is initialized by the bulk work items sent by the primary master (lines 287-289). The work items are sent one-by-one to the workers that communicate with the secondary masters. The image parts/results that are sent back, are forwarded to the primary master (lines 296 and 316).
- The secondary master nodes -expect to- receive a new work item for each result they send back. The new work item is appended to the local pool (line 320), so it is the next one to be sent to a worker. The worker assignment is delayed, until the work item in question is received from the master node (lines 301-307).

  Obviously, a proper, circular, FIFO queue setup would be more beneficial for the handling of each local work item pool. To avoid making an already lengthy code even longer, this is not pursued.
- The worker node code is identical to the one reported in Section 5.22, with the exception of line 347.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  #include <math.h>
5  #include "mpi.h"
6  #include <QImage>
7  #include <QRgb>
8  using namespace std;
9
10 //************************************************************
11 // Communication tags
12 #define NULLRESULTTAG 0
13 #define RESULTTAG      1
14 #define WORKITEMTAG    2
15 #define ENDTAG         3
16
17 #define BATCHSIZE 10              /* Jobs are initially send to ↩
       secondary master nodes in groups of BATCHSIZE
18                                    * BATCHSIZE has to be greater ↩
                                        than 1, otherwise the ↩
                                        secondary master nodes
19                                    * terminate prematurely
20                                    */
```

```
#define SECONDARYMASTER 2          /* Number of secondary master ↩
    nodes */
//**************************************************************

typedef struct WorkItem
{
  double upperX, upperY, lowerX, lowerY;
  int pixelsX, pixelsY, imageX, imageY, imgPartID;
} WorkItem;

//**************************************************************
// Class for computing a fractal set part
class MandelCompute
{
private:
  double upperX, upperY, lowerX, lowerY;
  int pixelsX, pixelsY, imgID;
  int *img;

  static int MAXITER;
  int diverge (double cx, double cy);

public:
    MandelCompute ();
  void init (WorkItem * wi);
   ~MandelCompute ();
  int *compute ();
};
int MandelCompute::MAXITER = 255;

//----------------------------------------

int MandelCompute::diverge (double cx, double cy)
{
  int iter = 0;
  double vx = cx, vy = cy, tx, ty;
  while (iter < MAXITER && (vx * vx + vy * vy) < 4)
    {
      tx = vx * vx - vy * vy + cx;
      ty = 2 * vx * vy + cy;
      vx = tx;
      vy = ty;
      iter++;
    }
  return iter;
}

//----------------------------------------
MandelCompute::~MandelCompute ()
{
  if (img != NULL)
    delete[] img;
}

//----------------------------------------
MandelCompute::MandelCompute ()
{
  img = NULL;
}

//----------------------------------------

void MandelCompute::init (WorkItem * wi)
{
  upperX = wi->upperX;
  upperY = wi->upperY;
  lowerX = wi->lowerX;
  lowerY = wi->lowerY;

  if (img == NULL || pixelsX != wi->pixelsX || pixelsY != wi->↩
      pixelsY)
    {
      if (img != NULL)
        delete[] img;
```

```cpp
      // img has an extra element for storing the ID of the picture part computed
      img = new int[(wi->pixelsX) * (wi->pixelsY) + 1];
    }
  pixelsX = wi->pixelsX;
  pixelsY = wi->pixelsY;
  imgID = wi->imgPartID;
}

//----------------------------------------

int *MandelCompute::compute ()
{
  double stepx = (lowerX - upperX) / pixelsX;
  double stepy = (upperY - lowerY) / pixelsY;

  for (int i = 0; i < pixelsX; i++)
    for (int j = 0; j < pixelsY; j++)
      {
        double tempx, tempy;
        tempx = upperX + i * stepx;
        tempy = upperY - j * stepy;
        img[j * pixelsX + i] = diverge (tempx, tempy);
      }
  img[pixelsX * pixelsY] = imgID;
  return img;
}

//**************************************************************
void registerWorkItem (MPI_Datatype * workItemType)
{
  struct WorkItem sample;

  int blklen[2];
  MPI_Aint displ[2], off, base;
  MPI_Datatype types[2];

  blklen[0] = 4;
  blklen[1] = 5;                 // the part's location is communicated regardless

  types[0] = MPI_DOUBLE;
  types[1] = MPI_INT;

  displ[0] = 0;
  MPI_Get_address (&(sample.upperX), &base);
  MPI_Get_address (&(sample.pixelsX), &off);
  displ[1] = off - base;

  MPI_Type_create_struct (2, blklen, displ, types, workItemType);
  MPI_Type_commit (workItemType);
}

//**************************************************************
// Uses the divergence iterations to pseudocolor the fractal set
void savePixels (QImage * img, int *imgPart, int imageX, int imageY, int height, int width)
{
  for (int i = 0; i < width; i++)
    for (int j = 0; j < height; j++)
      {
        int color = imgPart[j * width + i];
        img->setPixel (imageX + i, imageY + j, qRgb (256 - color, 256 - color, 256 - color));
      }
}

//**************************************************************
int main (int argc, char *argv[])
{
  int N, rank;
  double start_time, end_time;
  MPI_Status status;
  MPI_Request request;

  start_time = MPI_Wtime ();

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &N);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  if (N < SECONDARYMASTER * 2 + 1)
    {
      cerr << argv[0] << " needs at least " << SECONDARYMASTER * 2 + 1 << " nodes to run\n";
      MPI_Abort (MPI_COMM_WORLD, 2);
    }

  MPI_Datatype workItemType;
  registerWorkItem (&workItemType);

//----------------------------------
  if (rank == 0)                    // primary master code
    {
      if (argc < 6)
        {
          cerr << argv[0] << " upperCornerX upperCornerY lowerCornerX lowerCornerY workItemPixelsPerSide\n";
          MPI_Abort (MPI_COMM_WORLD, 1);
        }

      double upperCornerX, upperCornerY;
      double lowerCornerX, lowerCornerY;
      double partXSpan, partYSpan;
      int workItemPixelsPerSide;
      int Xparts, Yparts;
      int imgX = 1024, imgY = 768;

      upperCornerX = atof (argv[1]);
      upperCornerY = atof (argv[2]);
      lowerCornerX = atof (argv[3]);
      lowerCornerY = atof (argv[4]);
      workItemPixelsPerSide = atoi (argv[5]);

      // make sure that the image size is evenly divided in work items
      Xparts = (int) ceil (imgX * 1.0 / workItemPixelsPerSide);
      Yparts = (int) ceil (imgY * 1.0 / workItemPixelsPerSide);
      imgX = Xparts * workItemPixelsPerSide;
      imgY = Yparts * workItemPixelsPerSide;

      partXSpan = (lowerCornerX - upperCornerX) / Xparts;
      partYSpan = (upperCornerY - lowerCornerY) / Yparts;
      QImage *img = new QImage (imgX, imgY, QImage::Format_RGB32);

      // prepare the work items in individual structures
      WorkItem *w = new WorkItem[Xparts * Yparts];
      for (int i = 0; i < Xparts; i++)
        for (int j = 0; j < Yparts; j++)
          {
            int idx = j * Xparts + i;

            w[idx].upperX = upperCornerX + i * partXSpan;
            w[idx].upperY = upperCornerY - j * partYSpan;
            w[idx].lowerX = upperCornerX + (i + 1) * partXSpan;
            w[idx].lowerY = upperCornerY - (j + 1) * partYSpan;

            w[idx].imageX = i * workItemPixelsPerSide;
            w[idx].imageY = j * workItemPixelsPerSide;
            w[idx].pixelsX = workItemPixelsPerSide;
            w[idx].pixelsY = workItemPixelsPerSide;
            w[idx].imgPartID = idx;
          }

      // now distribute the work item to the worker nodes
      int *assignedPart = new int[SECONDARYMASTER];      // keep
```

```
          track of how many jobs each secondary master is ↩
          assigned
      int *imgPart = new int[workItemPixelsPerSide * ↩
          workItemPixelsPerSide + 1];        // for collecting ↩
          results
      int assignedCounter = 0;
      for (int i = 1; i <= SECONDARYMASTER; i++)
        {
          MPI_Recv (NULL, 0, MPI_INT, MPI_ANY_SOURCE, ↩
              NULLRESULTTAG, MPI_COMM_WORLD, &status);
          int secMasterID = status.MPI_SOURCE;
          int toBeAssigned = min (BATCHSIZE, Xparts * Yparts - ↩
              assignedCounter);
          assignedPart[secMasterID] = toBeAssigned;
          MPI_Isend (w + assignedCounter, toBeAssigned, ↩
              workItemType, secMasterID, WORKITEMTAG, ↩
              MPI_COMM_WORLD, &request);
          assignedCounter += toBeAssigned;
        }

      int remainPartToCollect = Xparts * Yparts;
      while (assignedCounter < Xparts * Yparts)
        {
          MPI_Recv (imgPart, workItemPixelsPerSide * ↩
              workItemPixelsPerSide + 1, MPI_INT, MPI_ANY_SOURCE↩
              , RESULTTAG, MPI_COMM_WORLD, &status);
          int secMasterID = status.MPI_SOURCE;
          int widx = imgPart[workItemPixelsPerSide * ↩
              workItemPixelsPerSide];    //assignedPart[workerID↩
              ];
          MPI_Isend (&(w[assignedCounter]), 1, workItemType, ↩
              secMasterID, WORKITEMTAG, MPI_COMM_WORLD, &request↩
              );

          savePixels (img, imgPart, w[widx].imageX, w[widx].↩
              imageY, workItemPixelsPerSide, ↩
              workItemPixelsPerSide);
          assignedCounter++;
          remainPartToCollect--;
        }

      while (remainPartToCollect > 0)
        {
          MPI_Recv (imgPart, workItemPixelsPerSide * ↩
              workItemPixelsPerSide + 1, MPI_INT, MPI_ANY_SOURCE↩
              , RESULTTAG, MPI_COMM_WORLD, &status);
          int secMasterID = status.MPI_SOURCE;
          int widx = imgPart[workItemPixelsPerSide * ↩
              workItemPixelsPerSide];
          MPI_Isend (NULL, 0, workItemType, secMasterID, ENDTAG, ↩
              MPI_COMM_WORLD, &request);

          savePixels (img, imgPart, w[widx].imageX, w[widx].↩
              imageY, workItemPixelsPerSide, ↩
              workItemPixelsPerSide);
          remainPartToCollect--;
        }


      img->save ("mandel.png", "PNG", 0);        // save the ↩
          resulting image

      delete[]w;
      delete[]assignedPart;
      delete[]imgPart;

      end_time = MPI_Wtime ();
      cout << "Total time : " << end_time - start_time << endl;
    }
//------------------------------------
  else if (rank <= SECONDARYMASTER)     // secondary master code
    {
      int workItemPixelsPerSide = atoi (argv[5]);
      WorkItem *localPool = new WorkItem[BATCHSIZE];
    int *imgPart = new int[workItemPixelsPerSide * workItemPixelsPerSide + 1];         // for collecting results
    int poolCount = 0;

    // get batch of load from primary master
    MPI_Send (NULL, 0, MPI_INT, 0, NULLRESULTTAG, MPI_COMM_WORLD);     // establish communication with primary master
    MPI_Recv (localPool, BATCHSIZE, workItemType, 0, WORKITEMTAG, MPI_COMM_WORLD, &status);
    MPI_Get_count (&status, workItemType, &poolCount);          // get the size of the assignment

    MPI_Request jobReq;
    bool waitFlag = false;
    int pendingResults = 0;
    while (poolCount > 0)
     {
      MPI_Recv (imgPart, workItemPixelsPerSide * workItemPixelsPerSide + 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int workerID = status.MPI_SOURCE;
      int tag = status.MPI_TAG;

      // wait for job(s) to be received from the MPI_Irecv call following below
      if (waitFlag)
       {
        MPI_Status st2;
        MPI_Wait (&jobReq, &st2);
        if (st2.MPI_TAG == WORKITEMTAG)
         poolCount++;
       }
      poolCount--;
      pendingResults++;
      MPI_Isend (localPool + poolCount, 1, workItemType, workerID, WORKITEMTAG, MPI_COMM_WORLD, &request);
      if (tag == RESULTTAG)
       {
        pendingResults--;

        // forward results to primary master
        MPI_Send (imgPart, workItemPixelsPerSide * workItemPixelsPerSide + 1, MPI_INT, 0, RESULTTAG, MPI_COMM_WORLD);

        // get new work item after the corresponding memory is free to accept new data
        MPI_Wait (&request, &status);
        MPI_Irecv (localPool + poolCount, 1, workItemType, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &jobReq);
        waitFlag = true;
       }
     }

    // now send termination messages
    while (pendingResults > 0)
     {
      MPI_Recv (imgPart, workItemPixelsPerSide * workItemPixelsPerSide + 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int workerID = status.MPI_SOURCE;
      int tag = status.MPI_TAG;

      MPI_Isend (NULL, 0, workItemType, workerID, ENDTAG, MPI_COMM_WORLD, &request);
      if (tag == RESULTTAG)
       {
        pendingResults--;

        // forward results to primary master
        MPI_Send (imgPart, workItemPixelsPerSide * workItemPixelsPerSide + 1, MPI_INT, 0,
                  RESULTTAG, MPI_COMM_WORLD);
            }
        }
      delete [] imgPart;
      delete [] localPool;
    }
//------------------------------------------
  else                              // worker code
    {
      int myMasterID = (rank % SECONDARYMASTER) + 1;     // which master to report to
      MandelCompute c;
      MPI_Send (NULL, 0, MPI_INT, myMasterID, NULLRESULTTAG, MPI_COMM_WORLD);   // establish communication with master
      while (1)
        {
          WorkItem w;
          MPI_Recv (&w, 1, workItemType, myMasterID, MPI_ANY_TAG, MPI_COMM_WORLD, &status);     // get a new work item
          int tag = status.MPI_TAG;
          if (tag == ENDTAG)
            break;

          c.init (&w);
          int *res = c.compute ();
          MPI_Send (res, w.pixelsX * w.pixelsY + 1, MPI_INT, myMasterID, RESULTTAG, MPI_COMM_WORLD);    // return the results
        }
    }

  MPI_Finalize ();
  return 0;
}
```

20. Modify the multi-threaded master-worker code of Section 5.22.2 so that there are two separate threads in each worker process for communication : one for receiving work items and one for sending back results. What are the benefits and drawbacks of this arrangement?

**Answer**

The benefits are two-fold:

(a) The decoupled "sender" and "receiver" codes are simpler to write and maintain. The less complicated logic is evident, as a simple comparison to Listing 5.40 can attest.

(b) The two communication operations can proceed at different speeds, potentially improving the utilization of the computing resources.

The modifications required are limited to the part of the `main` function that deal with worker communications (lines 391-436 of Listing 5.40). The additional `ResultSender` class shown below, is used to spawn a thread that handles communication *to* the master node exclusively, leaving the main thread to handle communication *from* the master node.

In terms of data items, the only change affects the `assigned` counter, which now needs to be changed from two threads. Instead of using a mutex to control access to it, its type is changed to `QAtomicInt`.

The following listing contains only the changes/additions to Listing 5.40, in order to reduce clutter.

```
...
class ResultSender:public QThread
{
private:
  QueueMonitor < MandelResult * >*outque;
  int numWorkerThreads;
  QAtomicInt *assigned;
public:
  ResultSender (QueueMonitor < MandelResult * >*oq, int nw, ↩
      QAtomicInt * as):outque (oq), numWorkerThreads (nw), ↩
      assigned (as)  {}
  void run ();
};

//*************************************************************
void ResultSender::run ()
{
  while (numWorkerThreads)
    {
      MandelResult *res;
      if (outque->availItems () > 0)
        {
          res = outque->deque ();
          if (res == NULL)
            {
              numWorkerThreads--;
            }
          else
            {
              MPI_Request r;
              MPI_Status s;

              MPI_Isend (res->getResultAddress (), res->↩
                  getResultSize (), MandelResult::type, 0, ↩
                  RESULTTAG, MPI_COMM_WORLD, &r);          // ↩
                  return the results
              MPI_Wait (&r, &s);

              outque->release (res);
              assigned->fetchAndAddOrdered (-1);
            }
        }
    }
}

//*************************************************************
int main (int argc, char *argv[])
{
...
    }
  else                              // worker code
    {
...
      // one loop for sending and recv messages
      bool endOfWork = false;
      int numWorkerThreads = numCores;
      QAtomicInt assigned (0);
      // spawn the thread for sending results back
      ResultSender rs (outque, numWorkerThreads, &assigned);
      rs.start ();

      // job receiving part
      while (!endOfWork)
        {
          MandelWorkItem *w = inque->reserve ();
          MPI_Recv (w, 1, MandelWorkItem::type, 0, MPI_ANY_TAG, ↩
              MPI_COMM_WORLD, &status);        // get a new work ↩
              item
          int tag = status.MPI_TAG;
          if (tag == ENDTAG)
            {
              for (int i = 0; i < numCores; i++)
                inque->enque (NULL);
              endOfWork = true;
            }
          else
            {
              inque->enque (w);
              assigned.fetchAndAddOrdered (1);
            }
        }

      for (int i = 0; i < numCores; i++)
        thr[i]->wait ();
      rs.wait ();
...
```

21. Conway's Game of Life is played on a rectangular grid of cells that may or may not contain an organism. The state of the cells is updated at each time step by applying the following set of rules:

    - Every organism with 2-3 neighbors survives.
    - Every organism with 4 or more neighbors dies from overpopulation.
    - Every organism with 0 - 1 neighbors dies from isolation.
    - Every empty cell adjacent to 3 organisms gives birth to a new one.

    Create an MPI program that evolves a board of arbitrary size (dimensions could be specified at the command-line) over several iterations. The board could be randomly generated or read from a file.

    Try applying the geometric decomposition pattern to partition the work among your processes. One example could be to evenly split the board row-wise. It is clear that each process can update its part of the board only by knowing the state of the bottom board row resident in the previous process, and the state of the top board row resident in the next process (the boundary processes being an exception).

    **Answer**

    The following solution uses geometric decomposition to distribute the board evenly among the MPI processes, in a row-wise fashion. At the end of each time step, the processes communicate their top and bottom rows with their neighbors, to allow proper calculation of neighbor cell counts. The arrangement is similar to the one used in Exercise 15, for the DLA problem. The difference with the depiction shown in Figure 5.2 is that only one instead of two rows is communicated per message.

```
// Game of Life
// G. Barlas, Dec. 2014
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h>

using namespace std;

#define PIC_SIZE 100
#define PARTICLES 500
#define MAX_ITER 10

#define DUMPIMAGES
/*------------------------------------------------------------*/
int countNeighbors (unsigned char *pic, int cols, int x, int y)
{
  int count = - pic[cols * y + x];
  for (int i = -1; i < 2; i++)
    for (int j = -1; j < 2; j++)
      count += pic[cols * (y + j) + (x + i)];

  return count;
}

/*------------------------------------------------------*/
/* Initializes the 2D array for the simulation */
void board_init (unsigned char *pic, int rows, int cols, int ↩
    particles)
{
  int i, j, x, y;
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
      pic[i * cols + j] = false;


  for (i = 0; i < particles; i++)       /* generate initial ↩
      particle placement */
    {
      x = random () % (cols - 2) + 1;   // counting starts from 1
      y = random () % (rows - 2) + 1;
      pic[y * cols + x] = true;
    }
}

/*------------------------------------------------------
 * Single step evolution of the simulation.
*/
void board_evolve (unsigned char *pic, unsigned char *pic2, int ↩
    rows, int cols)
{
  // prepare array to hold new state
  memset (pic2, 0, sizeof (unsigned char) * rows * cols);

  for (int y = 1; y < rows - 1; y++)
    for (int x = 1; x < cols - 1; x++)
      {
        int idx = y * cols + x;
        int c = countNeighbors (pic, cols, x, y);
        if (pic[idx])           // cell in-place
          {
            switch (c)
              {
// Cases are commented out because pic2 is reset already
//        case 0:
//        case 1:  // death - loniless
//          pic2[idx]= false;
//          break;
              case 2:
              case 3:           // survival
                pic2[idx] = true;
                break;
              default:          // death - overcrowded
//          pic2[idx]= false;
                break;
              }
          }
        else if (c == 3)        // birth in empty cell
          pic2[idx] = true;
      }
}

/*------------------------------------------------------*/
void saveImage (unsigned char * pic, int rows, int cols, char *↩
    fname)
{
  FILE *f = fopen (fname, "w+t");
  fprintf (f, "P1\n%i %i\n", cols, rows);

  for (int y = 1; y <= rows; y++)
    {
      for (int x = 1; x <= cols; x++)
```

```
        {
          if (pic[y * (cols + 2) + x])
            fprintf (f, "1 ");
          else
            fprintf (f, "0 ");
        }
      fprintf (f, "\n");
    }
  fclose (f);
}

/*-------------------------------------------------*/
int main (int argc, char **argv)
{
  int cols, rows, iter, particles, x, y;
  unsigned char *pic, *pic2, *tmp;
  int rank, num, i;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Request req, req2, req3;

  if (argc < 2)      // use default values if user does not specify anything
    {
      cols = PIC_SIZE;
      rows = PIC_SIZE;
      iter = MAX_ITER;
      particles = PARTICLES;
    }
  else
    {
      cols = atoi (argv[1]);
      rows = atoi (argv[2]);
      particles = atoi (argv[3]);
      iter = atoi (argv[4]);
    }

  // to make code simpler and avoid special cases, rows is forced to be a multiple of
  // the number of processes
  rows = (int) ceil (rows * 1.0 / num) * num;
  int rowsPerNode = rows / num;

  srand (time (0)); // initialize the random number generator

  // grid has two extra rows, one above and one below the actual rows
  // The extra rows "belong" to other nodes, so at the end of each time step
  // they should be exchanged
  if (rank == 0)
    {
      pic = new unsigned char[(cols + 2) * (rows + 2)];
      pic2 = new unsigned char[(cols + 2) * (rows + 2)];
    }
  else
    {
      pic = new unsigned char[(cols + 2) * (rowsPerNode + 2)];
      pic2 = new unsigned char[(cols + 2) * (rowsPerNode + 2)];
    }

  if (rank == 0)
    board_init (pic2, rows + 2, cols + 2, particles);
  // clean pic buffer
  memset (pic, 0, (cols + 2) * (rowsPerNode + 2) * sizeof (unsigned char));

  // boundary rows do not need to be communicated now.
  MPI_Scatter (pic2 + cols + 2, (cols + 2) * rowsPerNode, MPI_UNSIGNED_CHAR, pic + cols + 2, (cols + 2) * rowsPerNode, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

  int nextID = rank + 1, prevID = rank - 1;
  int step = 0;
  while (++step < iter)
    {
#ifdef DUMPIMAGES
      MPI_Gather (pic + cols + 2, rowsPerNode * (cols + 2), MPI_UNSIGNED_CHAR, pic + cols + 2, rowsPerNode * (cols + 2), MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
      if (rank == 0)
        {
          char fname[20];
          sprintf (fname, "%05i.pbm", step);
          saveImage (pic, rows, cols, fname);
        }
#endif

      board_evolve (pic, pic2, rowsPerNode + 2, cols + 2);
      tmp = pic2;
      pic2 = pic;
      pic = tmp;
      //exchange information with "previous" node
      if (rank != 0)
        {
          MPI_Isend (pic + cols + 2, cols + 2, MPI_UNSIGNED_CHAR, prevID, step, MPI_COMM_WORLD, &req);
          MPI_Irecv (pic, cols + 2, MPI_UNSIGNED_CHAR, prevID, step, MPI_COMM_WORLD, &req2);
        }

      // exchange with "next"
      if (rank != num - 1)
        {
          MPI_Isend (pic + rowsPerNode * (cols + 2), cols + 2, MPI_UNSIGNED_CHAR, nextID, step, MPI_COMM_WORLD, &req);
          MPI_Irecv (pic + (rowsPerNode + 1) * (cols + 2), cols + 2, MPI_UNSIGNED_CHAR, nextID, step, MPI_COMM_WORLD, &req3);
        }

      // block until messages are received
      if (rank != 0)
          MPI_Wait(&req2, MPI_STATUS_IGNORE);
      if (rank != num - 1)
          MPI_Wait(&req3, MPI_STATUS_IGNORE);
    }

  MPI_Gather (pic + cols + 2, rowsPerNode * (cols + 2), MPI_UNSIGNED_CHAR, pic + cols + 2, rowsPerNode * (cols + 2), MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
  if (rank == 0)
    saveImage (pic, rows, cols, "out.pbm");

  MPI_Finalize ();
  delete [] pic;
  delete [] pic2;
  return 0;
}
```

22. Radix sort is a linear complexity non-comparison-based sorting algorithm that is susceptible to concurrent execution. Radix sort sorts data by separating them into groups based on their digits (for integers) or characters (for strings). The data must be of fixed range, i.e. the number of bits or characters used must be known a priori.

    Radix sort comes in two forms: Least-Significant Digit radix sort (LSD) or Most-Significant Digit radix sort (MSD). The latter is suitable for parallel execution, as data which are partitioned in groups can be operated independently in subsequent phases of the algorithm. The MSD algorithm, which is very close to bucket-sort [2], can be implemented recursively as shown in the following pseudocode, for binary data. An extension for data with non-binary digits (strings) is straightforward. The use of the auxiliary array B allows the sorting to be *stable*.

```
// Input: array A, with N elements, each D bits long.
// Output : A holds sorted data
radix_sort(A, N)
   allocate memory B equal in size to A
   tmp <- radix_aux(A, N, B, D-1)
   if tmp <> A   // if sorted data end up in B array
     copy B to A

// Auxiliary recursive function
// Use of temporary array B for a stable sort
// Returns location of sorted data
radix_aux( A, N, B, k)
   if k = -1  Or N<2    // base case for termination
      return A
   else
      let r be the number of items in A, with k-th bit set to 0
      resetIdx <- 0
      setIdx <- r
      for i <- 0 to N-1    // separate data in two bins
         if k-th bit of A[i] is set
            store A[i] in B[ setIdx ]
            setIdx <- setIdx + 1
         else
            store A[i] in B[ resetIdx ]
            resetIdx <- resetIdx + 1

      tmp1 <- radix_aux(B, r, A, k-1)        // sort bin with a 0 ↩
          k-th bit, using the (k-1)-th bit
      tmp2 <- radix_aux(B+r, N-r, A+r, k-1) // sort bin with a 1 ↩
          k-th bit, using the (k-1)-th bit
      if tmp1 + r <> tmp2     // pointer comparison
          if r > N - r    // make the smallest copy
             copy N-r elements from tmp2 to tmp1+r
             return tmp1
          else
             copy r elements from tmp1 to tmp2-r
             return tmp2-r
      else
         return tmp1
```

Use divide & conquer decomposition as your decomposition pattern (see Section 2.3.2) to design and implement an MPI radix sort.

**Answer**

The algorithm design is influenced by the initial distribution of data, in the same way that bucketsort does. For example, it could be possible that all the nodes get a portion of the initial data, sort it according to one of the bits and exchange the resulting two parts. In this solution we consider the alternative where the data are originally resident at the 0-ranked node, which also performs the first sorting phase according to the most significant bit of the data values.

The following listing implements the divide & conquer decomposition pattern, whereas the data after each pass/phase of the algorithm that splits them into two groups, are split between two nodes. The overall communication pattern resembles a scattering operation. Node 0 first splits the

[2]The major difference between radix sort and bucket sort is in how the keys are examined: piece-wise in radix sort and as a whole in bucket sort. In the latter, an arbitrary number of buckets can be prescribed.

input into two pieces according to the value of the most significant bit, keeps one part for itself and sends the other part to node 1. During the second pass/phase, node 0 splits the remaining data and sends one of the parts to node 2, while node 1 performs the same operation for its part of the data and sends one of the parts to node 3. The process is repeated as many times as the number of bits used by the data values, but the migration of data to other nodes is limited by the number of MPI processes (check of line 120).

The key points of the following listing are:

- The data are initialized in node 0, and are sorted by having all the nodes call the `radixsort` function in line 195. The `radixsort` function sets-up the auxiliary memory needed for a stable sort (line 171) and calls the recursive function `radixSort_aux` which performs the actual sorting.
- In `radixSort_aux` each node determines the phase at which it will join the sorting operation (line 84), as well as the node from which it will receive its assignment (line 108), and the nodes to which it will subsequently assign parts of its workload (line 118).
- As with the quicksort algorithm, the split of parts is not even. In the event that a node does not have data to operate on (base case `M<=0` of line 88), it terminates, but not before informing all its sub-ordinate nodes that they should terminate as well (loop of lines 94 to 99).
- The `radixSort_aux` behaves differently during its first invocation, which is flagged by setting the `phase` parameter to -1. Namely, data are received from another node (line 109), and sent back to it following the completion of the "local" sort operation (line 153).
- The auxiliary array pointed to by `aux` is used to hold the data during the split operation carried out by function `radixPhaseK`. Data are shifted between the two memory locations (`data` and `aux`) during the different phases, with `radixSort_aux` maintaining a pointer (`tmp1`) to the actual location. The data should eventually return to their original location. If for any reason (e.g. an odd number of bits is used in the sorting) this is not done, the check in line 174 restores them.
- The `radixSort_aux` function also makes sure that the two data portions produced by the application of `radixPhaseK` in line 117, end up in the same location after they are independently sorted (lines 120-128 for the "left" part and line 131 for the "right" part). The necessary move is minimized by the logic of lines 134-148.

```
1   #include<mpi.h>
2   #include<string.h>
3   #include<stdio.h>
4   #include<stdlib.h>
5   #include<iostream>
6   #include<math.h>
7   #include<unistd.h>
8
9   using namespace std;
10
11  #define ASSIGNTAG 0
```

```
#define SORTEDRESTAG 1

const int MINV = 0;
const int MAXV = 10000;

//*****************************************
// first bit is at position 1
int msb (int i)
{
  int res = 0;
  while (i)
    {
      i >>= 1;
      res++;
    }
  return res;
}

//*****************************************
// Auxiliary function that performs a phase
// of a radixsort algorithm, based on the
// k-th bit. Temporary array B is used for
// a stable sort.
// Returns location the location where the
// sorted data "split apart". So B[r] is the
// first element with a 1-bit at position k
int radixPhaseK (int *A, int N, int *B, int k)
{
  if ((k == -1) || (N < 2))      // base case for termination
    {
      memcpy (B, A, N * sizeof (int)); // data are expected to ↩
          move to B
      return 0;
    }
  else
    {
      int r = 0;
      int bitMask = (1 << k);
      for (int i = 0; i < N; i++)
        if ((A[i] & bitMask) == 0)
          r++;

      int resetIdx = 0, setIdx = r;
      for (int i = 0; i < N; i++)
        {
          if ((A[i] & bitMask) > 0)
            {
              B[setIdx] = A[i];
              setIdx++;
            }
          else
            {
              B[resetIdx] = A[i];
              resetIdx++;
            }
        }
      return r;
    }
}

//*****************************************
// Recursive radisort function. Based on the
// phase, data maybe sent to other nodes for
// the subsequent phases.
// totalBits counts the bits left for comparison
// phase==-1 flags the first call to this function
int *radixSort_aux (int *data, int M, int *aux, int totalBits, ↩
    int rank, int num, int phase = -1)
{
  int newPhase;
  int sourceID;
  int *tmp1, *tmp2;
  if (phase == -1)                // if first call to recursive ↩
      function
```

```
   {
    newPhase = msb (rank);
    totalBits -= newPhase;    // corresponds to when the node will join the sorting
   }

  if (totalBits <= 0 || M <= 0) // base cases, no more sorting to be done
   {
    // have to terminate any nodes waiting for data
    if (phase != -1)
     {
      int partnerID = rank ^ (1 << phase);
      while (partnerID < num)
       {
        MPI_Send (NULL, 0, MPI_INT, partnerID, ASSIGNTAG, MPI_COMM_WORLD);
        phase++;
        partnerID = rank ^ (1 << phase);
       }
     }
    return data;
   }

  // first get data assignment
  MPI_Status st;
  if (phase == -1 && rank != 0)
   {
    sourceID = rank ^ (1 << (newPhase - 1));
    MPI_Recv (data, M, MPI_INT, sourceID, ASSIGNTAG, MPI_COMM_WORLD, &st);
    MPI_Get_count (&st, MPI_INT, &M); // get how many where actually received
   }

  if (phase != -1)
   newPhase = phase;

  // now split the data in two groups based on a bit value
  int split = radixPhaseK (data, M, aux, totalBits - 1);
  int partnerID = rank ^ (1 << newPhase);
  //either send one part to another node
  if (partnerID < num)
   {
    MPI_Send (aux, split, MPI_INT, partnerID, ASSIGNTAG, MPI_COMM_WORLD);
   }
  // or sort it locally
  else
   {
    tmp1 = radixSort_aux (aux, split, data, totalBits - 1, rank, num, newPhase + 1);
   }


  tmp2 = radixSort_aux (aux + split, M - split, data + split, totalBits - 1, rank, num, newPhase + 1);

  // make sure all data end-up at the same place, pointed by tmp1
  if (partnerID < num)     // no need to move data if the other part has been sent off
   tmp1 = tmp2 - split;
  else if (tmp2 != tmp1 + split)
   {
    // minimize movement by moving the smaller part
    if (split > M / 2)
     {
      memcpy (tmp1 + split, tmp2, (M - split) * sizeof (int));
     }
    else
     {
      memcpy (tmp2 - split, tmp1, split * sizeof (int));
      tmp1 = tmp2;
```

```
        }
    }

  // collect if data were sent away
  if (partnerID < num)
    {
      MPI_Recv (tmp1, split, MPI_INT, partnerID, SORTEDRESTAG, ↩
          MPI_COMM_WORLD, &st);
    }

  // before recursion ends, send back to source
  if (phase == -1 && rank != 0)
    MPI_Send (tmp1, M, MPI_INT, sourceID, SORTEDRESTAG, ↩
        MPI_COMM_WORLD);

  return tmp1;
}

//*****************************************
void radixSort (int *data, int M, int totalBits)
{
  int rank, num;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);

  int myPhase = msb (rank);
  int *aux = new int[M];

  int *tmp = radixSort_aux (data, M, aux, totalBits, rank, num);
  if (tmp != data)
    memcpy (data, tmp, M * sizeof (int));
  delete[] aux;
}


//*****************************************

int main (int argc, char **argv)
{
  MPI_Init (&argc, &argv);

  int rank, num, totalBits;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Status status;
  int M = atoi (argv[1]);        // get size of data to sort
  int *data;                     // data buffer

  data = new int[M];

  totalBits = ceil (log2 (MAXV));        // total number of phases↩
       for radix sort

  // initialize data array in rank 0
  if (rank == 0)
    {
      srand (time (0));
      for (int i = 0; i < M; i++)
        {
          data[i] = (rand () % (MAXV - MINV)) + MINV;
        }
    }

  radixSort (data, M, totalBits);

  // correctness check
  if (rank == 0)
    {
      for (int i = 1; i < M; i++)
        if (data[i] < data[i - 1])
          cout << "ERROR\n";
    }

  MPI_Finalize ();
  delete[] data;
  return 0;
}
```

# Chapter 6

# GPU Programming

## Exercises

1. An array of type `float` elements is to be processed in a one-element-per-thread fashion by a GPU. Suggest an execution configuration for the following scenarios:

   (a) The array is 1-D and of size $N$. The target GPU has 8 SMs, each with 16 SPs.

   (b) The array is 2-D and of size $NxN$. The target GPU has 5 SMs, each with 48 SPs.

   For each of the above scenario, calculate what is the minimum size that $N$ should satisfy to make the GPU computation a desirable alternative to CPU computation.

   **Answer**

   The number of data items should be high enough to provide work for all SPs.

   (a) A total of $8 \cdot 16 = 128$ SPs require at least $N \geq 128$. Ideally, $N$ should be a multiple of 128. An execution configuration can be obtained with the following code:

   ```
   int block = 16 * 4;  // 64 threads, a multiple of 16
   int grid = ( ( N - 1 ) / block ) + 1;
   foo<<< grid, block>>>(...);
   ```

   (b) A total of $5 \cdot 48 = 240$ SPs require at least $N^2 \geq 240 \Rightarrow N \geq 15.5$. Ideally, $N^2$ should be a multiple of 240. An execution configuration can be obtained with the following code:

   ```
   dim3 block (16, 12); // 192 threads, a multiple of 48
   dim3 grid ( ( N - 1 ) / 16 + 1,  ( N - 1 ) / 12 + 1 );
   foo<<< grid, block>>>(...);
   ```

2. A reduction is an operation frequently encountered in a many algorithms: summing-up the elements of an array, finding the minimum, maximum, etc. One possible solution to a CUDA kernel that calculates the sum of an array would be:

```
__global__ void sum(float *in, float *out)
{
   __shared__ float localStore[];  // to speedup data access

   int globalID = threadIdx.x + blockIdx.x * blockDim.x;
   int localID = threadIdx.x;

   localStore[localID] = in[globalID];   // copy to shared memory
   for(int i=1; i< blockDim.x ; i*=2)
     {
        if(localID % (2*i) == 0)
          localStore[localID] += localStore[localID + i];
        __syncthreads ();
     }
   if(localID == 0)
      out[blockIdx.x] = localStore[0];
}
```

The above needs to be called multiple times, each time reducing the size of the data by the number of threads in a block. Data at each kernel launch should be multiples of the block size.

Analyze the above code in relation to the following criteria and how they reflect on the execution speed: thread divergence, memory coalescing, use of SPs within a warp. Suggest modifications that would improve the performance.

**Answer**

The shown kernel scores low in all the mentioned areas, i.e. thread divergence, memory coalescing, and use of SPs within a warp. Analytically, the issues can be identified as follows:

- Thread divergence : The if condition inside the `for`-loop causes threads to diverge. During the first iteration only the threads with an ID which is a multiple of 2 can execute the `if` code body. During the second iteration only the ones with an ID which is a multiple of 4 can do this, and so on.
- Memory coalescing : There is no regard to memory coalescing inside the `for` loop. In fact, as the loop progresses, the number of bank conflicts increases.
- SP utilization : As soon as the `for` loop starts executing, an exponentially growing number of SPs with each iteration, become inactive, as they do not satisfy the `if` condition.

The following program can produce the same result with only one kernel invocation, with maximum SP utilization and minimum thread divergence:

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

#define MAXVALUE 10000

//------------------------------------
void numberGen (int N, int max, int *store)
{
  int i;
  srand (time (0));
  for (i = 0; i < N; i++)
    rand () % max;
```

```
}

//----------------------------------------
__global__ void sum (int *d, int N, int *odds)
{
  extern __shared__ int count[];

  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  int numThr = blockDim.x * gridDim.x;
  int localID = threadIdx.x;
  count[localID] = 0;
  if (myID < N)
    {
      count[localID] = d[myID];
      int idx = myID + numThr;
      // partial sum
      while (idx < N)
      {
        count[localID] += d[idx];
        idx += numThr;
      }

      __syncthreads ();
      // block-wide partial sum calculation
      int step = 1;
      while (((localID | step) < blockDim.x) && ((localID & step)↩
           == 0))
        {
          count[localID] += count[localID | step];
          step *= 2;
          __syncthreads ();
        }

      // now the block-wide partial sum is in count[0]
      // add to global counter
      if (localID == 0)
          atomicAdd (odds, count[0]);
    }
}

//----------------------------------------
int main (int argc, char **argv)
{
  int N = atoi (argv[1]);

  int *ha, *hres, *da, *dres;   // host (h*) and device (d*) ↩
      pointers

  ha = new int[N];
  hres = new int[1];

  cudaMalloc ((void **) &da, sizeof (int) * N);
  cudaMalloc ((void **) &dres, sizeof (int) * 1);

  numberGen (N, MAXVALUE, ha);

  cudaMemcpy (da, ha, sizeof (int) * N, cudaMemcpyHostToDevice);
  cudaMemset (dres, 0, sizeof (int));

  int blockSize, gridSize;
  blockSize = 256;
  gridSize = 16;
  sum <<< gridSize, blockSize, blockSize * sizeof (int) >>> (da, ↩
      N, dres);

  cudaMemcpy (hres, dres, sizeof (int), cudaMemcpyDeviceToHost);

  printf ("%i \n", *hres);

  cudaFree ((void *) da);
  cudaFree ((void *) dres);
  delete[] ha;
  delete[] hres;
  cudaDeviceReset ();
```

```
85
86   return 0;
87 }
```

The kernel of lines 17-51 allocates shared memory in the form of one integer per block thread. This memory is dynamically allocated during the kernel invocation (line 74). Each thread initially spends its time calculating a partial sum by going over the input data with a stride equal to the total number of threads (`while` loop of lines 30-34). Subsequently, a block-wide partial sum is calculated by reducing the shared memory counters with the loop of lines 39-44. The resulting sum in `count[0]` is added to the global memory counter with an atomic operation.

3. The reduction operation discussed in the previous exercise, is a special case of the "scan" or prefix-sum operation that can be applied to the elements of a vector or list. In general, the operator applied can be any of summation, subtraction, minimum, maximum, etc.. Implement a CUDA kernel capable of performing a prefix-sum operation. The "Prefix-Sums and Their Applications" paper by Guy Blelloch, available at `http://www.cs.cmu.edu/~guyb/papers/Ble93.pdf`, is a wonderful resource for learning more on the topic.

**Answer**

The solution is an extension of the one given in the previous exercise. The only differences lie in how the initial partial result for each thread is initialized (lines 34-45), in the operator used to perform the reduction (lines 56-67 and 78-89), and in the atomic operation used to consolidate the block-wide partial results with the global memory result (lines 98-109).

This program can be easily extended to handle any prefix-sum operation by introducing another symbolic constant representing it (following the ones defined in lines 8-10), and appropriately augmenting the `switch` statements.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <limits.h>
4  #include <cuda.h>
5
6  #define MAXVALUE 10000
7
8  #define SUM_OP 0
9  #define MIN_OP 1
10 #define MAX_OP 2
11
12 #define min(A,B) ((A>B) ? B : A)
13 #define max(A,B) ((A<B) ? B : A)
14 //---------------------------------------------
15 void numberGen (int N, int max, int *store)
16 {
17   int i;
18   srand (time (0));
19   for (i = 0; i < N; i++)
20     rand () % max;
21 }
22
23 //---------------------------------------------
24
25 __global__ void reduce (int *d, int N, int *res, int OP)
26 {
27   extern __shared__ int partRes[];
28
```

```
  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  int numThr = blockDim.x * gridDim.x;
  int localID = threadIdx.x;

  // initialize all the threads' storage regardless
  switch (OP)
    {
    case SUM_OP:
      partRes[localID] = 0;
      break;
    case MIN_OP:
      partRes[localID] = INT_MAX;
      break;
    case MAX_OP:
      partRes[localID] = INT_MIN;
      break;
    };

  if (myID < N)
    {
      partRes[localID] = d[myID];
      int idx = myID + numThr;

      // partial result
      while (idx < N)
        {
          int tmp = d[idx];
          switch (OP)
            {
            case SUM_OP:
              partRes[localID] += tmp;
              break;
            case MIN_OP:
              partRes[localID] = min (partRes[localID], tmp);
              break;
            case MAX_OP:
              partRes[localID] = max (partRes[localID], tmp);
              break;
            };
          idx += numThr;
        }

      __syncthreads ();
      // block-wide partial result calculation
      int step = 1;
      int otherIdx = localID | step;
      while ((otherIdx < blockDim.x) && ((localID & step) == 0))
        {
          int tmp = partRes[otherIdx];
          switch (OP)
            {
            case SUM_OP:
              partRes[localID] += tmp;
              break;
            case MIN_OP:
              partRes[localID] = min (partRes[localID], tmp);
              break;
            case MAX_OP:
              partRes[localID] = max (partRes[localID], tmp);
              break;
            };
          step <<= 1;
          otherIdx = localID | step;
          __syncthreads ();
        }

      // now the block-wide partial sum is in partRes[0]
      // add to global counter
      if (localID == 0)
        switch (OP)
          {
          case SUM_OP:
            atomicAdd (res, partRes[0]);
            break;
          case MIN_OP:
            atomicMin (res, partRes[0]);
            break;
          case MAX_OP:
            atomicMax (res, partRes[0]);
            break;
          };
    }
}

//------------------------------------
int sharedSize (int b)
{
  return b * sizeof (int);
}

//------------------------------------

int main (int argc, char **argv)
{
  int N = atoi (argv[1]);

  int *ha, *hres, *da, *dres;    // host (h*) and device (d*) pointers

  ha = new int[N];
  hres = new int[1];

  cudaMalloc ((void **) &da, sizeof (int) * N);
  cudaMalloc ((void **) &dres, sizeof (int) * 1);

  numberGen (N, MAXVALUE, ha);

  cudaMemcpy (da, ha, sizeof (int) * N, cudaMemcpyHostToDevice);
  cudaMemset (dres, 0, sizeof (int));

  int blockSize, gridSize;
  blockSize = 256;
  gridSize = 16;
  reduce <<< gridSize, blockSize, blockSize * sizeof (int) >>> (da, N, dres, MAX_OP);

  cudaMemcpy (hres, dres, sizeof (int), cudaMemcpyDeviceToHost);

  printf ("%i \n", *hres);

  cudaFree ((void *) da);
  cudaFree ((void *) dres);
  delete[] ha;
  delete[] hres;
  cudaDeviceReset ();

  return 0;
}
```

The code shown here is not nearly as optimum as it could be; it is a trade-off between readability and performance. A more thorough derivation of an optimum CUDA reduction kernel is covered in http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, although the material in question targets a Compute Capability 1.0 device, which exhibits many more quirks than contemporary architectures.

4. Create CUDA implementations of the `gsl_stats_mean()` and `gsl_stats_variance()` functions offered by the GNU Scientific Library, that produce the mean and variance statistics of an array of type `double` data. Their signatures are:

```
double gsl_stats_mean (const double DATA[], // Pointer to input data
                              size_t STRIDE,          // Step used to read ↩
                                  the input. Normally this should be ↩
                                  set to 1.
                              size_t N);              // Size of DATA array
double gsl_stats_variance (const double DATA[], // Same as above.
                                  size_t STRIDE,
                                  size_t N);
```

Assuming that the STRIDE is 1, create a memory access pattern that utilizes coalescing. Suggest ways to deal with the problem if the stride is not 1.

**Answer**

The `gsl_stats_mean` and `gsl_stats_variance` functions are implemented as host front-ends, to allow for the hiding of the implementation details (such as memory management and grid/block design) from the application that calls them.

The solution is based on the reduction approach that is used in the two previous exercises. Two kernels are employed for calculating the sum of values and the sum of the square of the values respectively, in order for the variance to be calculated from the formula:

$$var = \frac{\sum_{i=0}^{N-1} v_i^2}{N} - \left(\frac{\sum_{i=0}^{N-1} v_i}{N}\right)^2 \tag{6.1}$$

where $N$ is the number of values.

In order to deal with a STRIDE parameter which is different from 1, the `gsl_stats_mean` and `gsl_stats_variance` front-end functions, copy the data to be actually processed in another array (lines 127-130 and 169-172). The benefits are two-fold:

- A considerably smaller volume of data cross the PCIe bus, significantly reducing the communication cost.
- The data items reside in contiguous memory locations allowing for memory coalescing to take place, at least to the extent allowed by the `double` data type.

The last major point of the following listing is the `atomicAdd` device function of lines 22-34, that is an operation for `double`-type operands that is missing from the arsenal of CUDA primitives.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <cuda.h>
4
5   #define MAXVALUE 10000
6   #define min(A,B) ((A>B) ? B : A)
7   //-------------------------------------------
8   void numberGen (int N, int max, double *store)
9   {
10    int i;
11    srand (time (0));
12    for (i = 0; i < N; i++)
13      store[i] = i;                  //rand () % max;
14  }
15
16  //-------------------------------------------
17  int sharedSize (int b)
```

```
{
  return b * sizeof (double);
}
//-------------------------------------
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull =
                                            (unsigned long long int↩
                                                *)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                        __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
//-------------------------------------

__global__ void sum (double *d, int N, double *totalSum)
{
  extern __shared__ double partialSum[];

  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  int numThr = blockDim.x * gridDim.x;
  int localID = threadIdx.x;
  partialSum[localID] = 0;
  if (myID < N)
    {
      partialSum[localID] = d[myID];
      int idx = myID + numThr;
      // partial sum
      while (idx < N)
        {
          partialSum[localID] += d[idx];
          idx += numThr;
        }

      __syncthreads ();

      // reduction
      int step = 1;
      int otherIdx = localID | step;
      while ((otherIdx < blockDim.x) && ((localID & step) == 0))
        {
          partialSum[localID] += partialSum[otherIdx];
          step <<= 1;
          otherIdx = localID | step;
          __syncthreads ();
        }

      // now the block-wide partial sum is in partialSum[0]
      // add to global counter
      if (localID == 0)
        atomicAdd (totalSum, partialSum[0]);
    }
}

//-------------------------------------

__global__ void sum2 (double *d, int N, double *totalSum)
{
  extern __shared__ double partialSum[];

  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  int numThr = blockDim.x * gridDim.x;
  int localID = threadIdx.x;
  partialSum[localID] = 0;
  if (myID < N)
    {
      partialSum[localID] = d[myID]*d[myID];
      int idx = myID + numThr;
      // partial sum
      while (idx < N)
        {
          partialSum[localID] += d[idx]*d[idx];
          idx += numThr;
        }

      __syncthreads ();

      // reduction
      int step = 1;
      int otherIdx = localID | step;
      while ((otherIdx < blockDim.x) && ((localID & step) == 0))
        {
          partialSum[localID] += partialSum[otherIdx];
          step <<= 1;
          otherIdx = localID | step;
          __syncthreads ();
        }

      // now the block-wide partial sum is in partialSum[0]
      // add to global counter
      if (localID == 0)
        atomicAdd (totalSum, partialSum[0]);
    }
}

//----------------------------------------

double gsl_stats_mean (const double DATA[], size_t STRIDE, size_t↩
     N)
{
  double *ha, *da, *dres;   // host (h*) and device (d*) pointers
  double hres;

  int Neffective;
  if(STRIDE !=1)
  {
     Neffective = (N-1)/STRIDE + 1;
     ha = new double[Neffective];
     for(int i=0, j=0;i<N;i+=STRIDE, j++)
       ha[j] = DATA[i];
  }
  else
  {
    ha = (double *)DATA;
    Neffective = N;
  }

  cudaMalloc ((void **) &da, sizeof (double) * Neffective);
  cudaMalloc ((void **) &dres, sizeof (double) * 1);

  cudaMemcpy (da, ha, sizeof (double) * Neffective, ↩
      cudaMemcpyHostToDevice);
  cudaMemset (dres, 0, sizeof (double));

  int blockSize, gridSize;
  cudaOccupancyMaxPotentialBlockSizeVariableSMem (&gridSize, &↩
      blockSize, (void *) sum, sharedSize, Neffective);
  gridSize = min(gridSize, Neffective/blockSize);

  sum <<< gridSize, blockSize, blockSize * sizeof (double) >>> (↩
      da, Neffective, dres);

  cudaMemcpy (&hres, dres, sizeof (double), ↩
      cudaMemcpyDeviceToHost);

  if(STRIDE !=1)
     delete[]ha;

  cudaFree ((void *) da);
  cudaFree ((void *) dres);
  return hres/Neffective;
}
//----------------------------------------
```

```

double gsl_stats_variance (const double DATA[], size_t STRIDE, size_t N)
{
  double *ha, *da, *dres;   // host (h*) and device (d*) pointers
  double hsum, hsum2;

  int Neffective;
  if(STRIDE !=1)
  {
     Neffective = (N-1)/STRIDE + 1;
     ha = new double[Neffective];
     for(int i=0, j=0;i<N;i+=STRIDE, j++)
       ha[j] = DATA[i];
  }
  else
  {
    ha = (double *)DATA;
    Neffective = N;
  }

  cudaMalloc ((void **) &da, sizeof (double) * Neffective);
  cudaMalloc ((void **) &dres, sizeof (double) * 1);

  cudaMemcpy (da, ha, sizeof (double) * Neffective, cudaMemcpyHostToDevice);
  cudaMemset (dres, 0, sizeof (double));

  // first calculate the sum of values
  int blockSize, gridSize;
  cudaOccupancyMaxPotentialBlockSizeVariableSMem (&gridSize, &blockSize, (void *) sum, sharedSize, Neffective);

  gridSize = min(gridSize, Neffective/blockSize);
  sum <<< gridSize, blockSize, blockSize * sizeof (double) >>> (da, Neffective, dres);

  cudaMemcpy (&hsum, dres, sizeof (double), cudaMemcpyDeviceToHost);

  // then calculate the sum of squares
  cudaOccupancyMaxPotentialBlockSizeVariableSMem (&gridSize, &blockSize, (void *) sum2, sharedSize, Neffective);
  gridSize = min(gridSize, Neffective/blockSize);

  // reset the result variable
  cudaMemset (dres, 0, sizeof (double));
  sum2 <<< gridSize, blockSize, blockSize * sizeof (double) >>> (da, Neffective, dres);

  cudaMemcpy (&hsum2, dres, sizeof (double), cudaMemcpyDeviceToHost);

  if(STRIDE !=1)
     delete [] ha;

  cudaFree ((void *) da);
  cudaFree ((void *) dres);
  double avg = hsum/Neffective;
  return hsum2/Neffective - avg*avg;
}

//------------------------------------------

int main (int argc, char **argv)
{
  int N = atoi (argv[1]);

  double *data;

  data = new double[N];

  numberGen (N, MAXVALUE, data);

  printf ("Avg : %lf \n",gsl_stats_mean(data, 1, N));
  printf ("Var : %lf \n",gsl_stats_variance(data, 1, N) );

  delete [] data;
  cudaDeviceReset ();

  return 0;
}
```

5. Design and implement a CUDA program for calculating the histogram of a 24-bit color image. In this case, three separate histograms will be produced, one for each color component of the image.

   **Answer**

   The histogram-calculating kernels of Section 6.7.3 can be used as a basis for the solution. The only thing that needs to be explicitly addressed, is the handling of the three color components. Sending the raw RGB data to the GPU could result in having to launch just one kernel only. However, the memory layout in that case would be equivalent to an array of structures, which as discussed in Section 6.7.4 should be avoided.

   The alternative, is to rearrange the pixel data in color planes, as shown in lines 84-110, packing four pixel values per integer (line 106). This arrangement also permits the use of the histogram kernel presented in Listing 6.22, without any modifications.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <math.h>
5   #include <cuda.h>
6   #include <QImage>
7   #include <QRgb>
8
9   const int BINS = 256;
10  const int BINS4ALL = BINS * 32;
11
12  //↩
        ****************************************************************↩
13  __global__ void GPU_histogram_atomic (int *in, int N, int *h)
14  {
15    int gloID = blockIdx.x * blockDim.x + threadIdx.x;
16    int locID = threadIdx.x;
17    int GRIDSIZE = gridDim.x * blockDim.x;
18    __shared__ int localH[BINS4ALL];
19    int bankID = locID % warpSize;
20    int i;
21
22    // initialize the local, shared-memory bins
23    for (i = locID; i < BINS4ALL; i += blockDim.x)
24      localH[i] = 0;
25
26    // wait for all warps to complete the previous step
27    __syncthreads ();
28
29    //start processing the image data
30    int *mySharedBank = localH + bankID;
31    if (blockDim.x > warpSize) // if the blocksize exceeds the ↩
          warpSize, it is possible multiple warps run at the same ↩
          time
32      for (i = gloID; i < N; i += GRIDSIZE)
33        {
34
35          int temp = in[i];
36          int v = temp & 0xFF;
37          int v2 = (temp >> 8) & 0xFF;
```

```
        int v3 = (temp >> 16) & 0xFF;
        int v4 = (temp >> 24) & 0xFF;
        atomicAdd (mySharedBank + (v << 5), 1);
        atomicAdd (mySharedBank + (v2 << 5), 1);
        atomicAdd (mySharedBank + (v3 << 5), 1);
        atomicAdd (mySharedBank + (v4 << 5), 1);
      }
  else
    for (i = gloID; i < N; i += GRIDSIZE)
      {

        int temp = in[i];
        int v = temp & 0xFF;
        int v2 = (temp >> 8) & 0xFF;
        int v3 = (temp >> 16) & 0xFF;
        int v4 = (temp >> 24) & 0xFF;
        mySharedBank[v << 5]++;  // Optimized version of localH[↩
            bankID + v * warpSize]++
        mySharedBank[v2 << 5]++;
        mySharedBank[v3 << 5]++;
        mySharedBank[v4 << 5]++;
      }

  // wait for all warps to complete the local calculations, ↩
      before updating the global counts
  __syncthreads ();

  // use atomic operations to add the local findings to the ↩
      global memory bins
  for (i = locID; i < BINS4ALL; i += blockDim.x)
    atomicAdd (h + (i >> 5), localH[i]); // Optimized version of ↩
        atomicAdd (h + (i/warpSize), localH[i]);
}

//↩
    ****************************************************************↩
void histogramRGB_FE(QImage &img, int *hist)
{
    unsigned int *d_in, *h_in;
    int *d_hist;
    int N, Nx, Ny;

    Nx = img.width();
    Ny = img.height();
    N = ceil ((Nx * Ny) / 4.0);

    h_in = new unsigned int[N];
    cudaMalloc ((void **) &d_in, sizeof (int) * N);
    cudaMalloc ((void **) &d_hist, sizeof (int) * BINS);

    int gridSize=16, blockSize=256;
    for(int color=0;color<3;color++)
    {
        int idx=0;
        for(int y=0;y<Ny;y++)
          for(int x=0;x<Nx;x+=4)
          {
              unsigned int fourPixels=0;
              for(int i=0;i<4;i++)
              {
                  unsigned int tmp;
                  switch(color)
                  {
                  case 0:
                      tmp = qRed(img.pixel(x+i,y));
                      break;
                  case 1:
                      tmp = qGreen(img.pixel(x+i,y));
                      break;
                  default:
                      tmp = qBlue(img.pixel(x+i,y));
                      break;
                  }
```

```
                fourPixels = (fourPixels<<8) | tmp;
              }
            h_in[idx] = fourPixels;
            idx++;
          }

        cudaMemcpy (d_in, h_in, sizeof (int) * N, cudaMemcpyHostToDevice);
        cudaMemset (d_hist, 0, BINS * sizeof (int));

        GPU_histogram_atomic <<< gridSize, blockSize >>> ((int *)d_in, N, d_hist);

        cudaMemcpy (hist+color*BINS, d_hist, sizeof (int) * BINS, cudaMemcpyDeviceToHost);
    }

    cudaFree ((void *) d_in);
    cudaFree ((void *) d_hist);
    delete [] h_in;
}

//*****************************************************************
int main (int argc, char **argv)
{

  QImage pic;
  pic.load(argv[1]);

  int *hist = new int[3*BINS];
  histogramRGB_FE(pic, hist);

  for(int i=0;i<3;i++)
  {
      for(int j=0;j<10;j++)
          printf("%i ", hist[BINS*i+j]);
      printf(" ...\n");
  }
  cudaDeviceReset ();
  delete[]hist;
  return 0;
}
```

The above program, which relies on Qt for image I/O, can be compiled with:

```
$ nvcc -Xcompiler -fPIC -m64 -O2 -D_REENTRANT \
     -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB
     -I/usr/lib/x86_64-linux-gnu/qt5/mkspecs/linux-g++-64
     -I.
     -I/usr/include/qt5
     -I/usr/include/qt5/QtGui
     -I/usr/include/qt5/QtCore
     -lQt5Core -lQt5Gui
     -arch=sm_21
     -o histogram_atomic histogram_atomic.cu
```

6. Create a variation of the program in Listing 6.24 to discover and plot the memory copy speed for host-to-device and device-to-host operations and for all the available types of host memory allocations : pageable, pinned and mapped. In order to test the last type of allocation, you will have to call a kernel that will try to access the memory, triggering the transfer. Compare your results with the ones returned by the `$CUDA/samples/1_Utilities/bandwidthTest` sample program.

   **Answer**

This is an exercise that reveals a great deal about the underpinnings of the zero-copy memory mechanism. The program shown below transfers increasing amounts of data from the host to the device, using pageable, pinned and mapped memory allocation. In the last case, a dummy kernel is invoked to trigger the transfer of memory. The program parameters are:

(a) The number of iterations for calculating averages.

(b) The increment (in bytes) to use in testing increasing data chunks.

(c) The maximum data size (in bytes) to test for.

The maximum device memory is allocated in one go, as dictated by the third command-line parameter, in lines 59-66. The for loop of lines 68 to 72, calls the `timeCopyH2D` function once for every type of device memory and for increasing data sizes.

The `timeCopyH2D` function uses a stream and events in the same fashion as Listing 6.24, to calculate the average time taken to transfer data from the host to the device. When a zero-copy memory arrangement is used, the `dummyKernel` function is also invoked (line 33) in order to trigger the memory transfer instead of the explicit call of line 36.

```
 1 #include <stdio.h>
 2 #include <cuda.h>
 3 
 4 //-----------------------------------------------------------------
 5 // a dummy kernel just for initiating memory transfers when
 6 // mapped memory is used
 7 __global__ void dummyKernel (unsigned char *b, int N)
 8 {
 9   int idx = blockIdx.x * blockDim.x + threadIdx.x;
10   if (idx < N)
11     b[idx] = idx % 256;
12 }
13 
14 //-----------------------------------------------------------------
15 float timeCopyH2D (unsigned char *h, unsigned char *d, int N, int↩
       iter, bool ZEROCOPY = false)
16 {
17   cudaEvent_t startT, endT;
18   cudaStream_t str;
19   float duration;
20   int i;
21 
22   cudaEventCreate (&startT);
23   cudaEventCreate (&endT);
24   cudaStreamCreate (&str);
25 
26   cudaEventRecord (startT, str);
27   for (i = 0; i < iter; i++)
28     {
29       if (ZEROCOPY)
30         {
31           int block = 1024;
32           int grid = (N - 1) / block + 1;
33           dummyKernel <<< 1, 1, 0, str >>> (h, N);
34         }
35       else
36         cudaMemcpyAsync (d, h, N, cudaMemcpyHostToDevice, str);
37     }
38   cudaEventRecord (endT, str);
39   cudaEventSynchronize (endT);
40   cudaEventElapsedTime (&duration, startT, endT);
41 
42   cudaStreamDestroy (str);
43   cudaEventDestroy (startT);
44   cudaEventDestroy (endT);
45   return (duration / iter)/1000;  // convert to sec
46 }
47 
48 //-----------------------------------------------------------
49 int main (int argc, char **argv)
50 {
51   int iter = atoi (argv[1]);
52   int step = atoi (argv[2]);
53   int MAXDATASIZE = atoi (argv[3]);
54   unsigned char *h_data, *d_data;
55   unsigned char *h_pinned;
56   unsigned char *h_zerocopy;
57 
58   // pageable data allocation
59   h_data = (unsigned char *) malloc (MAXDATASIZE);
60   cudaMalloc ((void **) &d_data, MAXDATASIZE);
61 
62   //pinned host data allocation
63   cudaMallocHost ((void **) &h_pinned, MAXDATASIZE);
64 
65   // zero-copy data allocation
66   cudaHostAlloc ((void **) &h_zerocopy, MAXDATASIZE, cudaHostAllocMapped);
67   printf("SIZE PAGEABLE PINNED ZERO-COPY\n");
68   for (int dataSize = 0; dataSize <= MAXDATASIZE; dataSize += step)
69       printf ("%i %f %f %f\n", dataSize,
70                dataSize / timeCopyH2D (h_data, d_data, dataSize, iter) / (1<<20),
71                dataSize / timeCopyH2D (h_pinned, d_data, dataSize, iter)/ (1<<20),
72                dataSize / timeCopyH2D (h_zerocopy, d_data, dataSize, iter, true)/ (1<<20));
73 
74   cudaFreeHost (h_pinned);
75   cudaFreeHost (h_zerocopy);
76   free (h_data);
77   cudaFree (d_data);
78   cudaDeviceReset ();
79   return 1;
80 }
```

A sample of the results (in MB/sec) as tested on a GTX 870M GPU using the CUDA 6.5.14 SDK, are shown below:

```
$ ./memcpyTest 100 1000000 10000000
SIZE PAGEABLE PINNED ZERO-COPY
1000000 8721.193359 11407.150391 251645.046875
2000000 9870.787109 11431.939453 477716.156250
3000000 8816.846680 11335.143555 729376.500000
4000000 8428.890625 10872.275391 954590.687500
5000000 8996.895508 10910.654297 1273603.500000
6000000 9064.430664 11688.812500 1586918.125000
7000000 9636.149414 11648.382812 1916372.125000
8000000 9613.311523 11710.164062 2148302.250000
9000000 9656.351562 11678.595703 2497866.500000
10000000 9543.507812 11627.116211 2602141.250000
```

The results for pageable and pinned host memory are consistent with the speeds reported from the `$CUDA/samples/1_Utilities/bandwidthTest` utility. However, the results for the zero-copy memory are clearly bogus!

It seems that the issue is that the single thread launched in line 33 does not trigger the transfer of the entire designated data block. Forcing the invocation of the kernel for all the tested memory arrangements, results in the following `timeCopyH2D` function:

```
float timeCopyH2D (unsigned char *h, unsigned char *d, int N, int iter, bool ZEROCOPY = false)
{
  cudaEvent_t startT, endT;
  cudaStream_t str;
  float duration;
  int i;

  cudaEventCreate (&startT);
  cudaEventCreate (&endT);
  cudaStreamCreate (&str);

  cudaEventRecord (startT, str);
  for (i = 0; i < iter; i++)
    {
      int block = 1024;
      int grid = (N - 1) / block + 1;
      if (!ZEROCOPY)
        {
          cudaMemcpyAsync (d, h, N, cudaMemcpyHostToDevice, str);
          dummyKernel <<< grid, block, 0, str >>> (d, N);
        }
      else
        dummyKernel <<< grid, block, 0, str >>> (h, N);
    }
  cudaEventRecord (endT, str);
  cudaEventSynchronize (endT);
  cudaEventElapsedTime (&duration, startT, endT);

  cudaStreamDestroy (str);
  cudaEventDestroy (startT);
  cudaEventDestroy (endT);
  return (duration / iter)/1000;
}
```

The results of the modified program are shown below:

```
$ ./memcpyTest_withKernel 100 1000000 10000000
SIZE PAGEABLE PINNED ZERO-COPY
1000000 7139.290039 7236.488770 4724.393066
2000000 7622.321777 7693.185547 5189.917480
3000000 7627.702148 8057.083984 5141.410156
4000000 6630.308105 7898.557129 4420.788086
5000000 7482.463379 8091.921387 4954.079102
6000000 7526.963379 8168.914062 5034.622070
7000000 7304.780273 8191.183105 4979.723633
8000000 7429.434570 8235.093750 4981.714844
9000000 7413.144531 8203.048828 4856.591797
10000000 7374.308594 8240.675781 4991.039062
```

The above clearly illustrate that zero-copy memory is a recommended solution only if just part of the host data are required for the computation. Otherwise, even pageable memory is preferable in terms of performance. The delays introduced by the suspension of the threads that access the memory accumulate to a substantial performance hit.

7. The Mandelbrot set calculators of Section 6.12.1 are limited to a maximum of 255 iterations per pixel. However, the beauty of the Mandelbrot set is revealed for thousands or millions of iterations. Modify one or more of the solution of Section 6.12.1 so that up to $2^{16}-1$ iterations can be performed for each pixel. Profile your program and analyze its performance. What is the grid/block arrangement that yields the best performance?

   **Answer**

   The modifications required are minimum as they are concentrated on the memory management part of the code. Using the #1 version of the solution described in Section 6.12.1.1 as the basis of the answer, we need to modify only the `kernel.cu` file of Listing 6.29, so that the pixel values are stored as short integers (instead of unsigned characters). The changes to Listing 6.29 are documented below:

```
// Line 5, changing the upper iteration limit
static const int MAXITER = 0xFFFF;

// Line 26, changing the signature of the kernel to accommodate ↩
    the different type of pixel storage
__global__ void mandelKernel (unsigned short int *d_res, double ↩
    upperX, double upperY, double stepX, double stepY, int resX,↩
     int resY, int pitch)

// Lines 52, 53
unsigned short *h_res;
unsigned short *d_res;

// Lines 58, 59
CUDA_CHECK_RETURN (cudaMalloc ((void **) &d_res, resX * resY * ↩
    sizeof(unsigned short int)));
h_res = (unsigned short int *) malloc (resY * pitch * sizeof(↩
    unsigned short int));

// Line 71, getting the different-type results
CUDA_CHECK_RETURN (cudaMemcpy (h_res, d_res, resY * pitch * ↩
    sizeof(unsigned short int), cudaMemcpyDeviceToHost));
```

8. The stand-alone CUDA AES implementations of Section 6.12.2 suffer from a dominating data-transfer overhead, that exceeds the computational cost of the en-/de-cryption. Which of the following modifications will offer -if any- the biggest performance improvement?

   (a) *Use pinned host memory* : are there any concerns about the use of pinned memory for holding the whole of the data to be processed?

   (b) *Move the tables from constant to shared memory* : the constant memory cache is faster than global memory, but it is still slower than shared memory. Will the speed improvement offset the need to copy the tables to constant memory for every block of threads?

   (c) *Process multiple 16-byte blocks per thread* : turning `rijndaelGPUEncrypt()` into a `__device__` function and introducing another `__global__` function as a front-end to it, to be called by `rijndaelEncryptFE()`, should require the smallest possible effort.

   Modify the source code of the Version #2 program in order to introduce your chosen changes, and measure the performance improvement obtained.

   **Answer**

   (a) *Using pinned host memory* The problem with pinned memory is that it is a limited resource, and as such it would be prohibitive to store the entirety of the input data this way. Moving the input data to pinned memory in chunks prior to their transfer to the device would have no beneficial impact on the performance, as Version #2 already processes data in chunks of size `DEVICEMEMSIZE`, hence the piecewise transfer to pinned memory is already taking place implicitly.

   (b) *Moving the tables from constant to shared memory* The following prefix code is required in the `rijndaelGPUEncrypt` function contained in the `rijndael_device.cu` file, to copy the five encoding tables residing in constant memory to shared memory:

```
__global__ void rijndaelGPUEncrypt (int nrounds, u32 * data,↩
     int N)
{
  __shared__ u32 sTe0[256];
  __shared__ u32 sTe1[256];
  __shared__ u32 sTe2[256];
  __shared__ u32 sTe3[256];
  __shared__ u32 sTe4[256];

  for(int i=threadIdx.x; i< 256; i+= blockDim.x)
  {
    sTe0[i] = Te0[i];
    sTe1[i] = Te1[i];
    sTe2[i] = Te2[i];
    sTe3[i] = Te3[i];
    sTe4[i] = Te4[i];
  }
  __syncthreads();
...
```

The performance improvement is dramatic : as tested on a GTX 870M GPU using the CUDA 6.5.14 SDK, the encoding time of a 512MB file is reduced from 2.53s to 0.6s.

(c) *Processing multiple 16-byte blocks per thread* This modification can be achieved with fewer changes, by reducing the size of the grid and introducing a loop in the `rijndaelGPUEncrypt` function. The resulting code, combining the changes of the previous step, is shown below:

```
// changes to file rijndael_host_streams.cu
const int BLOCKSPERTHREAD=4;

void rijndaelEncryptFE (const u32 * rk, int keybits, ↩
    unsigned char *plaintext, unsigned char *ciphertext, int↩
     N, int thrPerBlock = 256)
{
 . . .
      // grid calculation
      int grid = ceil((numDataBlocks*1.0) / (thrPerBlock * ↩
          BLOCKSPERTHREAD));

      rijndaelGPUEncrypt <<< grid, thrPerBlock, 0, str[↩
          whichStream] >>> (nrounds, d_buffer[whichStream], ↩
          toSend);
 . . .



// changes to file rijndael_device.cu
__global__ void rijndaelGPUEncrypt (int nrounds, u32 * data,↩
     int N)
{
  __shared__ u32 sTe0[256];
  __shared__ u32 sTe1[256];
  __shared__ u32 sTe2[256];
  __shared__ u32 sTe3[256];
  __shared__ u32 sTe4[256];

  for (int i = threadIdx.x; i < 256; i += blockDim.x)
    {
      sTe0[i] = Te0[i];
      sTe1[i] = Te1[i];
      sTe2[i] = Te2[i];
      sTe3[i] = Te3[i];
      sTe4[i] = Te4[i];
    }
  __syncthreads ();

  int myID = blockIdx.x * blockDim.x + threadIdx.x;
  int totalThr = gridDim.x * blockDim.x;
  for (; myID < (N >> 4); myID += totalThr)
    {
      u32 s0, s1, s2, s3, t0, t1, t2, t3;
      const u32 *rk = d_rk;  // to avoid changing d_rk
      u32 aux;

      int myDataIdx = myID << 2; // *4 to offset 16 bytes

// code below this line is shared with the original ↩
    rijndaelGPUEncrypt function
#ifndef FULL_UNROLL
      int r;
. . .
      data[myDataIdx + 3] = RESHUFFLE (s3);
    }
}
```

In terms of performance, there is virtually no change as the encryption/decryption of a single block already constitutes a sufficiently heavy computational load.

9. The MPI cluster AES implementations of Section 6.12.2 does not provide overlapping of communication and computation. This issue could be addressed if a new "work item" was downloaded by the worker nodes while the GPU was processing an already downloaded part. Modify the MPI solution to provide this functionality. Do you expect any problems with the load-balancing of the modified solution?

   **Answer**

   A solution to this problem could be that each worker sets-up buffer space to hold three work items, and starts processing one while using a non-blocking send and a non-blocking receive operations to communicate the completed and next pending work items respectively.

   The only modifications required, relate to the worker code as shown below for the case of a GPU worker. An identical arrangement is possible for a CPU worker. The communication/computation overlap is achieved by using immediate communication primitives to exchange data with the master (lines 31, 36) while computation is taking place (line 38). There is no need to use immediate functions in lines 29 and 34, as scalar primitive types can be buffered by MPI :

```
1  // changes to file AES_MPI/main.cpp
2    if (rank == 0)
3     . . .
4     else                              // GPU worker
5      {
6        int workItemSize = atoi (argv[3]);
7        int thrPerBlock = atoi (argv[4]);
8        int pos[3] = { -1, -1, -1 };
9        int totalWork = 0;
10       unsigned char *workBuff[3];
11       workBuff[0] = new unsigned char[workItemSize];
12       workBuff[1] = new unsigned char[workItemSize];
13       workBuff[2] = new unsigned char[workItemSize];
14       int actualSize[3];
15       MPI_Request getRq, sendRq;
16
17       int activeBuff = 0;        // indices for work item buffer ↩
             management
18       int completedBuff = 2;
19       int nextBuff = 1;
20
21      MPI_Send (&(pos[completedBuff]), 1, MPI_INT, 0, TAG_RES, ↩
            MPI_COMM_WORLD);
22      MPI_Recv (&(pos[activeBuff]), 1, MPI_INT, 0, TAG_WORK, ↩
            MPI_COMM_WORLD, &stat);
23      MPI_Recv (workBuff[activeBuff], workItemSize, ↩
            MPI_UNSIGNED_CHAR, 0, TAG_DATA, MPI_COMM_WORLD, &stat)↩
            ;
24      MPI_Get_count (&stat, MPI_UNSIGNED_CHAR, &(actualSize[↩
            activeBuff]));
25      totalWork += actualSize[activeBuff];
26      while (pos[activeBuff] >= 0)
27        {
28          // send completed item with immediate send
29          MPI_Send (&(pos[completedBuff]), 1, MPI_INT, 0, TAG_RES↩
                , MPI_COMM_WORLD);
30          if (pos[completedBuff] > 0)
31            MPI_Isend (workBuff[completedBuff], actualSize[↩
                  completedBuff], MPI_UNSIGNED_CHAR, 0, TAG_DATA, ↩
                  MPI_COMM_WORLD, &sendRq);
32
33          // get next item while processing
34          MPI_Recv (&(pos[nextBuff]), 1, MPI_INT, 0, TAG_WORK, ↩
                MPI_COMM_WORLD, &stat);
35          if (pos[nextBuff] > 0)
36            MPI_Irecv (workBuff[nextBuff], workItemSize, ↩
                  MPI_UNSIGNED_CHAR, 0, TAG_DATA, MPI_COMM_WORLD, ↩
                  &getRq);
37
38          rijndaelEncryptFE (rk, keybits, workBuff[activeBuff], ↩
                workBuff[activeBuff], actualSize[activeBuff], ↩
                thrPerBlock);
39
40      // make sure communications are complete before updating ↩
            the indices and comntinuing
41          if (pos[completedBuff] > 0)
42            MPI_Wait (&sendRq, MPI_STATUS_IGNORE);
43          if (pos[nextBuff] > 0)
44            {
45              MPI_Wait (&getRq, &stat);
46              MPI_Get_count (&stat, MPI_UNSIGNED_CHAR, &(↩
                    actualSize[nextBuff]));
47              totalWork += actualSize[nextBuff];
48            }
49          // update indices
50          completedBuff = activeBuff;
51          activeBuff = nextBuff;
52          nextBuff = (nextBuff + 1) % 3;
53        }
54      rijndaelShutdown ();
55      cout << "Worker " << rank << " processed " << totalWork << ↩
            endl;
56      delete[] workBuff[0];
57      delete[] workBuff[1];
58      delete[] workBuff[2];
59    }
```

The three work item buffers are addressed individually by the `completedBuff`, `activeBuff` and `nextBuff` indices. Each buffer is accompanied by a dedicated position (`pos` array) and length (`actualSize` array) variables. While one buffer is being processed (pointed to by `activeBuff`), another holding previous results (pointed to by `completedBuff`) is being send to the master node (line 31) and the third one (pointed to by `nextBuff`) is being filled with new input data (line 36).

In all the communication transactions, the `pos` array members guard against sending dummy data to the master (line 30) and against waiting for new data when the input is exhausted (lines 26 and 35).

A problem may arise with the load balancing of this solution if a very slow worker is participating in the proceedings. It is possible that it can delay the termination of the overall execution by receiving a pending work item that could be processed faster at another node.

10. Modify the MPI cluster AES implementations of Section 6.12.2 so that only two types of messages are needed for data exchange, instead of the current three. How can this be combined with the modification of the previous exercise?

    **Answer**

    The messages tagged by the `TAG_WORK` label can be eliminated from the communication protocol if the master node maintains a record of which part of the data is assigned to each worker node respectively. When a single work item is assigned to each worker, a simple integer array with as many element as the number of workers, is sufficient for handling this requirement.

    ```
      int assignedWorkItem[comm_size];
    ```

    If more than a single work item is assigned at a time to a worker, then the `assignedWorkItem` array has to become an array of containers, such as a `vector<int>`:

    ```
    vector<int> assignedWorkItem[comm_size];
    ```

    The termination of the workers can be achieved by sending a work item of length zero.

    There is also another alternative. The whole point of this exercise is to *reduce the number of messages sent*, while at the same time being able to identify the origin/destination in the input/output data of the data parts being exchanged. Towards this end, we could utilize the tag of the messages to carry this information (it is of type `int`). The only drawback of collapsing the two messages into one is that there is a need for an extra data copy from the generic message buffer used to collect the incoming results (line 33) to the output data repository (line 38):

    ```
    1    if (rank == 0)
    2      {
    3        unsigned char *iobuf;
    4        unsigned char *recvbuf;
    5        if ((f = fopen (argv[1], "r")) == NULL)
    6          {
    7            fprintf (stderr, "Can't open %s\n", argv[1]);
    8            exit (EXIT_FAILURE);
    9          }
    10
    11       int workItemSize = atoi (argv[3]);
    12
    13       fseek (f, 0, SEEK_END);
    14       lSize = ftell (f);
    15       rewind (f);
    16
    17       iobuf = new unsigned char[lSize];
    18       recvbuf = new unsigned char[workItemSize];
    19       assert (iobuf != NULL);
    20       assert (recvbuf != NULL);
    21       fread (iobuf, 1, lSize, f);
    22       fclose (f);
    23
    24       timeval tim;
    25       gettimeofday (&tim, NULL);
    ```

```
      double tm2 = tim.tv_sec + (tim.tv_usec / 1000000.0);

      // master main loop
      int pos = 0;
      while (pos < lSize)
        {
          int retPos, recvSize;
          MPI_Recv (recvbuf, workItemSize, MPI_UNSIGNED_CHAR, ↩
              MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat↩
              );
          MPI_Get_count (&stat, MPI_UNSIGNED_CHAR, &recvSize);
          if (recvSize > 0)
            {
              retPos = stat.MPI_TAG;
              memcpy (iobuf + retPos, recvbuf, recvSize);
            }

          // assign next work item
          int actualSize = (workItemSize < lSize - pos) ? ↩
              workItemSize : (lSize - pos);
          MPI_Send (iobuf + pos, actualSize, MPI_UNSIGNED_CHAR, ↩
              stat.MPI_SOURCE, pos, MPI_COMM_WORLD);
          pos += actualSize;
        }

      // wait for last results
      for (int i = 1; i < comm_size; i++)
        {
          int retPos, recvSize;
          MPI_Recv (recvbuf, workItemSize, MPI_UNSIGNED_CHAR, ↩
              MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat↩
              );
          MPI_Get_count (&stat, MPI_UNSIGNED_CHAR, &recvSize);
          if (recvSize > 0)
            {
              retPos = stat.MPI_TAG;
              memcpy (iobuf + retPos, recvbuf, recvSize);
            }

          // send termination signal
          MPI_Send (recvbuf, 0, MPI_UNSIGNED_CHAR, stat.↩
              MPI_SOURCE, 0, MPI_COMM_WORLD);
        }
      gettimeofday (&tim, NULL);
      double tm3 = tim.tv_sec + (tim.tv_usec / 1000000.0);

      FILE *fout;
      if ((fout = fopen (argv[2], "w")) == NULL)
        {
          fprintf (stderr, "Can't open %s\n", argv[2]);
          exit (EXIT_FAILURE);
        }
      fwrite (iobuf, 1, lSize, fout);
      fclose (fout);
      delete[] iobuf;
      delete[] recvbuf;

      gettimeofday (&timeMain, NULL);
      double tm4 = timeMain.tv_sec + (timeMain.tv_usec / ↩
          1000000.0);

      // print-out some timing information
      printf ("%.9lf \t %.9lf \n", tm4 - tm1, tm3 - tm2);
    }
  else                          // GPU worker
    {
      int workItemSize = atoi (argv[3]);
      int thrPerBlock = atoi (argv[4]);
      int pos[3] = { 0, 0, 0 };
      int totalWork = 0;
      unsigned char *workBuff[3];
      workBuff[0] = new unsigned char[workItemSize];
      workBuff[1] = new unsigned char[workItemSize];
      workBuff[2] = new unsigned char[workItemSize];
```

```
92      int actualSize[3] = { 0, 0, 0 };
93      MPI_Request getRq, sendRq;
94
95      int activeBuff = 0;        // indices for work item buffer ↩
            management
96      int completedBuff = 2;
97      int nextBuff = 1;
98
99      MPI_Send (workBuff[completedBuff], 0, MPI_UNSIGNED_CHAR, 0,↩
             pos[completedBuff], MPI_COMM_WORLD);
100     MPI_Recv (workBuff[activeBuff], workItemSize, ↩
            MPI_UNSIGNED_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &↩
            stat);
101     pos[activeBuff] = stat.MPI_TAG;
102     MPI_Get_count (&stat, MPI_UNSIGNED_CHAR, &(actualSize[↩
            activeBuff]));
103     totalWork += actualSize[activeBuff];
104     while (actualSize[activeBuff] > 0)
105       {
106         // send completed item with immediate send
107         MPI_Isend (workBuff[completedBuff], actualSize[↩
                completedBuff], MPI_UNSIGNED_CHAR, 0, pos[↩
                completedBuff], MPI_COMM_WORLD, &sendRq);
108
109         // get next item while processing
110         MPI_Irecv (workBuff[nextBuff], workItemSize, ↩
                MPI_UNSIGNED_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD,↩
                 &getRq);
111
112         rijndaelEncryptFE (rk, keybits, workBuff[activeBuff], ↩
                workBuff[activeBuff], actualSize[activeBuff], ↩
                thrPerBlock);
113
114         // make sure communications are complete before ↩
                updating the indices and comntinuing
115         MPI_Wait (&sendRq, MPI_STATUS_IGNORE);
116         MPI_Wait (&getRq, &stat);
117         MPI_Get_count (&stat, MPI_UNSIGNED_CHAR, &(actualSize[↩
              nextBuff]));
118         pos[nextBuff] = stat.MPI_TAG;
119         totalWork += actualSize[nextBuff];
120
121         completedBuff = activeBuff;
122         activeBuff = nextBuff;
123         nextBuff = (nextBuff + 1) % 3;
124       }
125     rijndaelShutdown ();
126     cout << "Worker " << rank << " processed " << totalWork << ↩
            endl;
127     delete [] workBuff[0];
128     delete [] workBuff[1];
129     delete [] workBuff[2];
130   }
```

In the above code the explicit communication of the buffer positions is eliminated. Termination and first job request conditions are detected by checking the length of the input (line 35) and output data (line 104) communicated respectively.

11. The whole point of the multi-core "adventure", is to accelerate our programs. This should be our sole focus, beyond any mis- or pre-conceptions. The evaluation of the different AES parallel implementations conducted in Section 6.12.2.4 considered only the encryption process, disregarding any I/O costs incurred. Perform your own experiment where the overall execution time is considered and not just the encryption time. Make sure that the file cache provided by the operating system is not utilized by:

   - Either, calling the following from the command-line (root permissions are required):

```
$ sync ; echo 3 > /proc/sys/vm/drop_caches
```

- Or, calling the `posix_fadvice()` function from within your program, prior to any I/O:

```
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char *argv[]) {
  int fd;
  fd = open(argv[1], O_RDONLY); // Open the file holding the↩
       input data
  fdatasync(fd);
  posix_fadvise(fd, 0,0,POSIX_FADV_DONTNEED); // clear cache
  close(fd);
  . . .
}
```

Analyze your findings.

**Answer**

This is an activity that is to be carried out by the students. The following command line can be used for generating a random 32MB input file to be used in testing:

```
$ dd if=/dev/urandom of=in32MBytes bs=32M count=1
```

# Chapter 7

# The Thrust Template Library

## Exercises

1. Develop a Thrust program for calculating the inner product of two vectors by using the `thrust::transform` algorithm and an appropriate functor.

   **Answer**

   The `thrust::transform` algorithm can be used to multiple the elements of two vectors, but it has to be accompanied by the `thrust::reduce` algorithm in order to sum-up the partial products. In the following code, two randomly-initialized vectors, of user-supplied size (as specified by a single command-line parameter), are multiplied accordingly. The result is computed on both the CPU and GPU in order to compare the results:

```
#include <iostream>
#include <stdlib.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/transform.h>
#include <thrust/reduce.h>

using namespace std;
struct innerProdFunct
{
  __host__ __device__ float operator () (float &x, float &y)
  {
    return x * y;
  }
};

int main (int argc, char **argv)
{
  int N = atoi (argv[1]);
  thrust::host_vector < float >h_x (N);
  thrust::host_vector < float >h_y (N);
  thrust::host_vector < float >h_prod (N);
  thrust::device_vector < float >d_x;
  thrust::device_vector < float >d_y;
  thrust::device_vector < float >d_prod (N);

  srand (time (0));
  for (int i = 0; i < N; i++)
    {
      h_x[i] = rand () % 1000;
      h_y[i] = rand () % 1000;
    }

  // copy data to device
  d_x = h_x;
  d_y = h_y;
  // device-based calculation
  thrust::transform (d_x.begin (), d_x.end (), d_y.begin (), ↩
      d_prod.begin (), innerProdFunct ());
  float res = thrust::reduce (d_prod.begin (), d_prod.end ());
  cout << "Inner product on GPU is " << res << endl;

  // host calculation
  thrust::transform (h_x.begin (), h_x.end (), h_y.begin (), ↩
      h_prod.begin (), innerProdFunct ());
  res = thrust::reduce (h_prod.begin (), h_prod.end ());
  cout << "Inner product on CPU is " << res << endl;

  return 0;
}
```

2. Calculate the inner product of two vectors by using the `thrust::inner_product` algorithm. Compare the performance of this version with the performance of a CUDA-based solution.

   **Answer** The use of the `thrust::inner_product` can replace the sequence of `thrust::transform` and `thrust::reduce` of the previous exercise. The resulting program does not require a functor for the calculation to take place:

```
#include <iostream>
#include <stdlib.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/inner_product.h>

using namespace std;

int main (int argc, char **argv)
{
  int N = atoi (argv[1]);
  thrust::host_vector < float >h_x (N);
  thrust::host_vector < float >h_y (N);
  thrust::device_vector < float >d_x;
  thrust::device_vector < float >d_y;

  srand (time (0));
  for (int i = 0; i < N; i++)
    {
      h_x[i] = rand () % 1000;
      h_y[i] = rand () % 1000;
    }

  d_x = h_x;
  d_y = h_y;

  float res = thrust::inner_product (d_x.begin (), d_x.end (),
                                     d_y.begin (),
                                     0,          // initial reduction value
                                     thrust::plus < float >(),  // reduction
                                     thrust::multiplies < float >()); // ↩
                                         transformation
  cout << "Inner product function on GPU produces " << res << ↩
      endl;

  return 0;
}
```

3. Use Thrust algorithms to find the absolute maximum of an array of values.

**Answer**

The solution can be based on the `thrust::transform_reduce` algorithm, which permits kernel fusion. A custom unary functor (lines 8-14) can be used to calculate the absolute values during the transformation phase, with the built-in `thrust::maximum` functor used in the reduction phase.

```
 1  #include <iostream>
 2  #include <stdlib.h>
 3  #include <thrust/device_vector.h>
 4  #include <thrust/host_vector.h>
 5  #include <thrust/transform_reduce.h>
 6
 7  using namespace std;
 8  struct absFunct
 9  {
10    __host__ __device__ float operator () (float &x)
11    {
12      return fabs (x);
13    }
14  };
15
16  int main (int argc, char **argv)
17  {
18    int N = atoi (argv[1]);
19    thrust::host_vector < float >h_x (N);
20    thrust::device_vector < float >d_x;
21
22    srand (time (0));
23    for (int i = 0; i < N; i++)
24      {
25        h_x[i] = rand () % 1000;
26      }
27
28    d_x = h_x;
29    float max = thrust::transform_reduce (d_x.begin (), d_x.end (),
30                                          absFunct (),
31                                          0,
32                                          thrust::maximum < float ↩
                                                >());
33    cout << "Max by GPU: " << max << endl;
34
35
36    max = 0;
37    for (int i = 0; i < N; i++)
38      {
39        float temp = fabs (h_x[i]);
40        if (temp > max)
41          max = temp;
42      }
43    cout << "Max by CPU: " << max << endl;
44
45    return 0;
46  }
```

4. Write a Thrust program for calculating the sum of two matrices.

**Answer**

The solution can be based on the `thrust::transform` algorithm, combined with the built-in `thrust::plus` functor. The matrices can be simply treated as vectors, as their dimensions have no influence on the computation (as they do in multiplication). In the following code the user supplies the dimensions of the matrices to be randomly generated for testing the code.

A `thrust::host_vector` is used for initializing the two matrices (lines 39-40 and 43-44) before passing them to the device (lines 41 and 45) and for retrieving and printing out the result (lines 49-50).

```
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/random.h>
#include <math.h>

using namespace std;

void printMatrix (thrust::host_vector < float >&m, int N, int M)
{
  for (int i = 0; i < N; i++)
    {
      cout << "| ";
      for (int j = 0; j < M; j++)
        {
          cout << m[i * M + j] << " ";
        }
      cout << "|" << endl;
    }
  cout << "==============================\n";
}

//*****************************************************
int main (int argc, char **argv)
{
  // initialize the RNG
  thrust::default_random_engine rng (time (0));
  thrust::uniform_int_distribution < int >uniDistr (-10000, 10000);

  int N = atoi (argv[1]);
  int M = atoi (argv[2]);

  // generate the data on the host and move them to the device
  thrust::device_vector < float >x (N * M);
  thrust::device_vector < float >y (N * M);
  thrust::device_vector < float >z (N * M);
  thrust::host_vector < float >aux (N * M);
  for (int i = 0; i < x.size (); i++)
    aux[i] = uniDistr (rng);
  x = aux;
  printMatrix (aux, N, M);
  for (int i = 0; i < x.size (); i++)
    aux[i] = uniDistr (rng);
  y = aux;
  printMatrix (aux, N, M);

  thrust::transform (x.begin (), x.end (), y.begin (), z.begin (), thrust::plus < float >());
  aux = z;
  printMatrix (aux, N, M);

  return 0;
}
```

5. Write a Thrust program for calculating the definite integral of a function $f(x)$ over a range $[a, b]$, using the trapezoidal rule. Consider a solution that avoids the need to create a vector for all the x-values for which a trapezoid is calculated. The details of the trapezoidal rule can be found in Section 3.5.2.

   **Answer**

   The answer is to employ a `thrust::counting_iterator` to generate a sequence of indices, and a functor which scales them to generate $x$ values for the calculation of the desired function.

   The functor shown in the listing below (lines 13-28) is initialized with the appropriate offset (data member `a`) and scale (data member `h`) to convert an index $i$ to an appropriate $x_i$ value (line 25), as dictated by the equation that calculates the integral of a function $f()$ in the range $[a, b]$ :

$$h\left(\frac{f(a)+f(b)}{2}+\sum_{i=1}^{n-1} f(x_i)\right)$$

where $x_0 = a$, $x_n = b$, $x_i = a + i \cdot h$ and $h = \frac{b-a}{n}$.

Line 26 calculates the value of the function. The `thrust::transform_reduce` algorithm of lines 39-42, sums-up the produced function values, effectively calculating the summation term of the above equation.

By having the initial reduction value set to `( f(0.0f) + f(N * 1.0f) )/2`, (`f(0.0f)` evaluates to `f(a)` and `f(N * 1.0f)` to `f(b)`)the reduction calculates the complete parenthesis expression. The multiplication by $h$ in line 43 completes the result.

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <thrust/iterator/counting_iterator.h>
5  #include <thrust/transform_reduce.h>
6  #include <thrust/functional.h>
7
8  const float LOWERLIMIT = 0;  // stands for range limit a
9  const float UPPERLIMIT = 10; // stands for range limit b
10
11 using namespace std;
12 //*****************************************
13 struct integrFunct
14 {
15   float h, a;
16
17     integrFunct (float st, float width)
18   {
19     a = st;
20     h = width;
21   }
22
23   __host__ __device__ float operator () (float i)
24   {
25     float x = a + h * i;
26     return fabs (sin (x));
27   }
28 };
29 //*****************************************
30 int main (int argc, char **argv)
31 {
32   int N = atoi (argv[1]);
33
34   float h = (UPPERLIMIT - LOWERLIMIT) / N;;
35   integrFunct f (LOWERLIMIT, h);
36
37   thrust::counting_iterator < float >x (1);
38
39   float res = thrust::transform_reduce (x, x + N - 1,
40                                         f,
41                                         ( f(0.0f) + f(N * 1.0f) ) ↩
                                               /2,
42                                         thrust::plus < float >());
43   res *= h;
44   cout << "Definite integral is " << res << endl;
45
46   return 0;
47 }
```

6. Use Thrust to calculate the mean and variance of a data set $X$ of cardinality $N$. For convenience, the corresponding formulas are:

$$E[X] = \frac{\sum_{i=0}^{N-1} x_i}{N} \tag{7.1}$$

$$\sigma^2 = E[X^2] - (E[X])^2 \tag{7.2}$$

**Answer**

This is a straight-forward application of the `thrust::reduce` algorithm for calculating the sum of the data set values, and the `thrust::transform_reduce` algorithm for calculating the sum of the square data set values. The remaining calculations are very simple.

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/random.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>

using namespace std;
//*****************************************
struct sqrFunct
{
  __host__ __device__ float operator () (float x)
  {
    return x*x;
  }
};
//*****************************************
int main (int argc, char **argv)
{
  int N = atoi (argv[1]);  // initialize the RNG
  thrust::default_random_engine rng (time(0));
  thrust::uniform_int_distribution<int> uniDistr(-10000,10000);

  // generate the data on the host and move them to the device
  thrust::device_vector < float >d_x (N);
  thrust::host_vector<float> h_x(N);

  for (int i = 0; i < N; i++) h_x[i] = uniDistr(rng);
  d_x = h_x;

  // calculate the sum of values
  float avg = thrust::reduce(d_x.begin(), d_x.end());
  avg /= N;
  //calculate the sum of squared values
  float sum2 = thrust::transform_reduce (d_x.begin(), d_x.end(),
                                        sqrFunct(),
                                        0,
                                        thrust::plus < float >())↩
                                            ;

  cout << "Mean : " << avg << endl;
  cout << "Variance : " << sum2/N - avg*avg << endl;

  return 0;
}
```

7. Measure the performance of the `thrust::sort` algorithm in Thrust by sorting varying volumes of data. Compare the achieved times with the STL version running on the host. For this purpose create a big array (make sure it is not too big to fit in the GPU's memory) and populate it with random data.

**Answer**

The following listing employs the POSIX high-resolution timing functions (see Appendix C.2) for reporting the time it takes to sort a variable-size array of integers using STL on the host and Thrust on the device respectively. The program requires three command-line parameters corresponding to the minimum (`Nmin`) and maximum array size (`Nmax`), and the iteration step used in testing (`Nstep`).

Once the arrays are initialized and placed in their respective memory spaces, the loop of lines 40-52 proceeds to sort an increasing part of the arrays and report the time and processing rate in million integers/sec units.

```
1  #include <iostream>
2  #include <algorithm>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <thrust/device_vector.h>
6  #include <thrust/host_vector.h>
7  #include <thrust/sort.h>
8
9  using namespace std;
10
11 //************************************************
12
13 double hrclock_sec ()
14 {
15   timespec ts;
16   clock_gettime (CLOCK_REALTIME, &ts);
17   double aux = ts.tv_sec + ts.tv_nsec / 1000000000.0;
18   return aux;
19 }
20
21 //************************************************
22
23 int main (int argc, char **argv)
24 {
25   int Nmin = atoi (argv[1]);
26   int Nmax = atoi (argv[2]);
27   int Nstep = atoi (argv[3]);
28   thrust::host_vector < float >h_x (Nmax);
29   thrust::device_vector < float >d_x;
30
31   srand (time (0));
32   for (int i = 0; i < Nmax; i++)
33     {
34       h_x[i] = rand () % 1000000;
35     }
36
37   d_x = h_x;
38
39   double cpu_time, gpu_time, t0, t1, t2;
40   for (int items = Nmin; items <= Nmax; items += Nstep)
41     {
42       t0 = hrclock_sec ();
43       std::sort (h_x.begin (), h_x.begin () + items);
44       t1 = hrclock_sec ();
45       cpu_time = t1 - t0;
46
47       thrust::sort (d_x.begin (), d_x.begin () + items);
48       t2 = hrclock_sec ();
49       gpu_time = t2 - t1;
50
51       cout << items << " Times CPU,GPU (sec) : " << cpu_time << " " << gpu_time << "\t Rates (million integers/sec) : " << items / cpu_time / 1000000 << " " << items / gpu_time / 1000000 << "\n";
52     }
53
54   return 0;
}
```

The results of a sample run on a 3.5GHz i7-3770K CPU and a GTX 560Ti GPU are shown below:

```
$ nvcc −O2 sortThrust.cu − o sortThrust
$ ./sortThrust 1000000 5000000 500000
1000000 Times CPU,GPU (sec) : 0.0548186 0.00260711      Rates (←
    million integers/sec) : 18.242 383.567
1500000 Times CPU,GPU (sec) : 0.0649869 0.00337315      Rates (←
    million integers/sec) : 23.0816 444.689
2000000 Times CPU,GPU (sec) : 0.0791855 0.00413704      Rates (←
    million integers/sec) : 25.2572 483.438
2500000 Times CPU,GPU (sec) : 0.0953443 0.00491595      Rates (←
    million integers/sec) : 26.2208 508.548
3000000 Times CPU,GPU (sec) : 0.103896 0.00588679       Rates (←
    million integers/sec) : 28.8751 509.615
3500000 Times CPU,GPU (sec) : 0.124521 0.0065577        Rates (←
    million integers/sec) : 28.1077 533.723
4000000 Times CPU,GPU (sec) : 0.127966 0.00732589       Rates (←
    million integers/sec) : 31.2583 546.009
4500000 Times CPU,GPU (sec) : 0.136482 0.00801635       Rates (←
    million integers/sec) : 32.9713 561.353
5000000 Times CPU,GPU (sec) : 0.151194 0.00879908       Rates (←
    million integers/sec) : 33.0701 568.241
```

8. Measure the performance hit that an Array-of-Structures design approach will have in device-based sorting of $10^8$ randomly generated instances of the following structure:

```
struct TestStr {
  int key;
  float value;

  __host__ __device__
  bool operator<(const TestStr &o) const
  {
     return this.key < o.key;
  }
};
```

To measure the performance deterioration, you will have to implement and time a Structure-of-Arrays alternative.

**Answer**

The answer is given in the listing below. While the `thrust::sort` algorithm can used for sorting the array of structures, the `thrust::sort_by_key` algorithm needs to be employed for the structure-of-arrays alternative. The data are randomly generated and are common in both arrangements.

The POSIX high-resolution timing functions (see Appendix C.2) are used for reporting the time.

```
#include <iostream>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/sort.h>

using namespace std;

//**********************************************

double hrclock_sec ()
{
  timespec ts;
  clock_gettime (CLOCK_REALTIME, &ts);
  double aux = ts.tv_sec + ts.tv_nsec / 1000000000.0;
  return aux;
}

//************************************************
struct TestStr
{
  int key;
  float value;

  __host__ __device__ bool operator< (const TestStr & o) const
  {
    return this->key < o.key;
  }
};

//************************************************

int main (int argc, char **argv)
{
  int N = atoi (argv[1]);
  thrust::host_vector < TestStr > h_str (N);
  thrust::device_vector < TestStr > d_str;

  thrust::host_vector < int >h_keys (N);
  thrust::host_vector < float >h_values (N);
  thrust::device_vector < int >d_keys (N);
  thrust::device_vector < float >d_values (N);

  srand (time (0));
  for (int i = 0; i < N; i++)
    {
      h_str[i].key = rand () % 1000000;
      h_str[i].value = rand () % 1000000;

      h_keys[i] = h_str[i].key;
      h_values[i] = h_str[i].value;
    }

  d_str = h_str;
  d_keys = h_keys;
  d_values = d_values;

  double t0, t1, t2;
  t0 = hrclock_sec ();
  thrust::sort (d_str.begin (), d_str.end ());
  t1 = hrclock_sec ();
  thrust::sort_by_key (d_keys.begin (), d_keys.end (), d_values.↩
      begin ());
  t2 = hrclock_sec ();
  cout << N << " " << t1 - t0 << " " << t2 - t1 << endl;

  return 0;
}
```

9. Create a Thrust program for computing the Mandelbrot set. Section 3.8.2 covers the mathematical details of how the set is calculated.

   **Answer**

   Thrust has the limitation of not providing a 2D counting iterator, that would have been handy in this problem. We can still use the `thrust::counting_iterator` facility as long as we can couple it with a transformation that maps each generated index to a coordinate on the complex plane. In the functor of lines 19-57 in the listing that follows, this is exactly what is taking place.

   The functor is initialized with all the information necessary to map what is essentially a pixel counter running from top-to-bottom and left-to-right, into a pair of complex plane coordinates. The conversion is taking place in lines 40-43. The remaining lines in the `operator()` method is just a copy of the `MandelCompute::diverge` method shown in Listing 3.29 in page 127.

The `thrust::transform` call in line 85 calculates the number of iterations required for each examined complex plane point to diverge. These data are subsequently copied to the host (line 88), prior to using them for painting the pixels of a *QImage* object (lines 91-96).

```
 1  // To compile :
 2  // nvcc -m64 -O2 -Xcompiler '-fPIE' -D_REENTRANT -DQT_NO_DEBUG -↩
        DQT_GUI_LIB -DQT_CORE_LIB -I/usr/lib/x86_64-linux-gnu/qt5/↩
        mkspecs/linux-g++-64 -I. -I/usr/include/qt5 -I/usr/include/↩
        qt5/QtGui -I/usr/include/qt5/QtCore -I. -o mandelThrust ↩
        mandelThrust.cu -L /opt/cuda/lib64/ -lcudart -rdc=true -arch↩
        =sm_21 -lm  -lQt5Core -lQt5Gui -L /usr/lib/x86_64-linux-gnu/
 3
 4  #include <iostream>
 5  #include <stdlib.h>
 6  #include <math.h>
 7  #include <QImage>
 8  #include <QRgb>
 9  #include <thrust/device_vector.h>
10  #include <thrust/host_vector.h>
11  #include <thrust/iterator/counting_iterator.h>
12  #include <thrust/transform.h>
13
14  using namespace std;
15
16  const int MAXITER = 255;
17
18  //****************************************
19  struct mandelFunct
20  {
21    double upperX, upperY;
22    double xStep, yStep;
23    int imageWidth;
24
25    mandelFunct (double uX, double uY, double xS, double yS, int w)
26    {
27      upperX = uX;
28      upperY = uY;
29      xStep = xS;
30      yStep = yS;
31      imageWidth = w;
32    }
33
34    __host__ __device__ int operator  () (int i)
35    {
36      int xIDX, yIDX;
37      double cx, cy;
38      int iter = 0;
39
40      xIDX = i % imageWidth;
41      yIDX = i / imageWidth;
42      cx = upperX + xIDX * xStep;
43      cy = upperY - yIDX * yStep;
44
45      double vx = cx, vy = cy, tx, ty;
46      while (iter < MAXITER && (vx * vx + vy * vy) < 4)
47        {
48          tx = vx * vx - vy * vy + cx;
49          ty = 2 * vx * vy + cy;
50          vx = tx;
51          vy = ty;
52          iter++;
53        }
54      return iter;
55    }
56
};

//*****************************************
int main (int argc, char **argv)
{
  int imgX = 1024, imgY = 768;
  double upperCornerX, upperCornerY;
  double lowerCornerX, lowerCornerY;
  double xStep, yStep;

  upperCornerX = atof (argv[1]);
  upperCornerY = atof (argv[2]);
  lowerCornerX = atof (argv[3]);
  lowerCornerY = atof (argv[4]);

  xStep = (lowerCornerX - upperCornerX) / imgX;
  yStep = (upperCornerY - lowerCornerY) / imgY;

  QImage *img = new QImage (imgX, imgY, QImage::Format_RGB32);

  mandelFunct f (upperCornerX, upperCornerY, xStep, yStep, imgX);

  thrust::host_vector < int >h_iter;
  thrust::device_vector < int >d_iter (imgX * imgY);

  thrust::counting_iterator < float >x (0);

  // calculate the pixel values on the device
  thrust::transform (x, x + imgX * imgY, d_iter.begin (), f);

  // copy results to host
  h_iter = d_iter;

  // draw the image pixels accordingly
  for (int j = 0; j < imgY; j++)
    for (int i = 0; i < imgX; i++)
      {
        int color = h_iter[imgX * j + i];
        img->setPixel (i, j, qRgb (256 - color, 256 - color, 256 ↩
            - color));
      }
  img->save ("mandel.png", "PNG", 0);
  return 0;
}
```

10. A problem related to the furthest-distance point solved in Section 7.5, is the problem of finding the pair of points which are the furthest apart from each other. Create a brute-force solution to this problem, i.e. by examining all pairs of points, using Thrust. Consider the case where the number of points is too big to allow storage of all the calculated distances. You should avoid duplicate calculations since the distance from point A to B is the same as the distance from B to A.

    **Answer**

    Not being able to store the distances creates both opportunities and difficulties:

    - The number $N$ of points that can be used as input is not limited by the need to allocate $\Theta(\frac{N^2-N}{2})$ storage for the pair distances.
    - The distances might have to be calculated multiple times.

    Each pair of points can be identified by the index of the corresponding cell in the distances table. We can compare the "indices" of two pairs of points, by calculating and comparing the corresponding pair-wise distances. In order to avoid duplicate calculations, we can limit the indices to either of

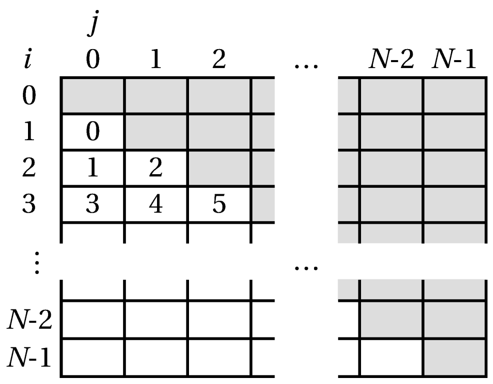


Figure 7.1: Illustration of how a pair of points and their distance can be identified by the index of the corresponding cell in the distances matrix. The indices are calculated by numbering the cells in a top-to-bottom and left-to-right fashion, skipping the gray cells. The matrix is not actually allocated.

the two table halves, as split by the main diagonal. In the following code we use the lower half. The indices are calculated as illustrated in Figure 7.1, i.e. to identify the pair of point $(i, j)$, with $i > j$, we use the formula:

$$idx_{(i,j)} = \sum_{k=1}^{i-1} k + j = \frac{i(i-1)}{2} + j \tag{7.3}$$

The opposite mapping, i.e. from an index to a pair of points can be also performed. Given an initial approximation for $i$ :

$$idx_{(i,j)} = \frac{i^2 - i)}{2} + j \Rightarrow i^2 = 2 \cdot idx_{(i,j)} - 2j + i \Rightarrow i \approx \sqrt{2 \cdot idx_{(i,j)}} \tag{7.4}$$

we can calculate an exact value for $i$ and $j$ by utilizing the knowledge that $i > j$:

```
i = (int) floor (sqrt (2.0 * idx));
while ((j = idx - i * (i - 1) / 2) >= i)
    i++;
```

The solution given below is based on the application of a `thrust::reduce` algorithm on a `thrust::counting_iterator` that enumerates all the possible cell indices/point pairs. Upon comparing two indices, the supplied functor (lines 20-49) reverses the mapping to convert the two pair indices into two pairs of points, pair `A` and pair `B`. The point indices `Ai`, `Aj` and `Bi`, `Bj` are subsequently used to calculate and compare the square of the distances $\|\overrightarrow{AiAj}\|^2$ $\|\overrightarrow{BiBj}\|^2$ (indices used liberally here to represent the actual points).

The only restriction of the following solution is that the type `thrust::counting_iterator<int>` iterator is appropriate for up to $N < 46340$. For bigger inputs, a switch to `thrust::counting_iterator<long>` would be necessary, as a 32-bit signed integer is not sufficient for holding the generated indices.

```
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/reduce.h>
#include <thrust/random.h>
#include <math.h>

using namespace std;

//*****************************************************
__host__ __device__ void reverseMap (int idx, int &i, int &j)
{
  i = (int) floor (sqrt (2.0 * idx));
  while ((j = idx - i * (i - 1) / 2) >= i)
    i++;
}

//*****************************************************
struct maxFunct
{
  thrust::device_ptr < int >x, y, z;

  // comparing distances of pairs of points A and B as indexed by↩
       idxA, idxB
  __host__ __device__ int operator  () (int idxA, int idxB)
  {
    int Ai, Aj, Bi, Bj;
    int distA, distB;

    reverseMap (idxA, Ai, Aj);
    reverseMap (idxB, Bi, Bj);

    int xDiff, yDiff, zDiff;
    xDiff = x[Ai] - x[Aj];
    yDiff = y[Ai] - y[Aj];
    zDiff = z[Ai] - z[Aj];
    distA = xDiff * xDiff + yDiff * yDiff + zDiff * zDiff;

    xDiff = x[Bi] - x[Bj];
    yDiff = y[Bi] - y[Bj];
    zDiff = z[Bi] - z[Bj];
    distB = xDiff * xDiff + yDiff * yDiff + zDiff * zDiff;

    if (distA > distB)
        return idxA;
    else
        return idxB;
  }
};

//*****************************************************
int main (int argc, char **argv)
{
  // initialize the RNG
  thrust::default_random_engine rng (time (0));
  thrust::uniform_int_distribution < int >uniDistr (-10000, ↩
      10000);

  int N = atoi (argv[1]);

  // generate the data on the host and move them to the device
  thrust::device_vector < int >d_x (N);
  thrust::device_vector < int >d_y (N);
  thrust::device_vector < int >d_z (N);
  thrust::host_vector < int >h_x (N);
  thrust::host_vector < int >h_y (N);
  thrust::host_vector < int >h_z (N);
  for (int i = 0; i < N; i++) h_x[i] = uniDistr (rng);
  d_x = h_x;
  for (int i = 0; i < N; i++) h_y[i] = uniDistr (rng);
  d_y = h_y;
  for (int i = 0; i < N; i++) h_z[i] = uniDistr (rng);
  d_z = h_z;

  // initialize the functor that will find the maximum distance, so that it has access to the coordinate arrays
  maxFunct f;
  f.x = d_x.data ();
  f.y = d_y.data ();
  f.z = d_z.data ();

  // reduce the index of the most distant point
  int furthest = thrust::reduce (thrust::counting_iterator < int >(0),
                                 thrust::counting_iterator < int >((N * N - N) / 2),
                                 0,
                                 f);

  cout << "The most distant pair is " << furthest << endl;
  int Ai, Aj;
  reverseMap (furthest, Ai, Aj);
  cout << "First point is " << Ai << "(" << h_x[Ai] << " " << h_y[Ai] << " " << h_z[Ai] << ")" << endl;
  cout << "Second point is " << Aj << "(" << h_x[Aj] << " " << h_y[Aj] << " " << h_z[Aj] << ")" << endl;

  // CPU verification
  double max_dist = 0, maxI, maxJ;
  for (int i = 0; i < N - 1; i++)
    for (int j = i + 1; j < N; j++)
      {
        int xDiff, yDiff, zDiff;
        xDiff = h_x[i] - h_x[j];
        yDiff = h_y[i] - h_y[j];
        zDiff = h_z[i] - h_z[j];
        int dist = xDiff * xDiff + yDiff * yDiff + zDiff * zDiff;
        if (dist > max_dist)
          {
            max_dist = dist;
            maxI = i;
            maxJ = j;
          }
      }
  cout << endl;
  cout << "Distance by CPU : " << max_dist << endl;
  cout << "First point is " << maxI << "(" << h_x[maxI] << " " << h_y[maxI] << " " << h_z[maxI] << ")" << endl;
  cout << "Second point is " << maxJ << "(" << h_x[maxJ] << " " << h_y[maxJ] << " " << h_z[maxJ] << ")" << endl;

  return 0;
}
```

11. The all-pairs-shortest paths graph problem can be solved by the dynamic programming algorithm by Floyd and Warshall. Assuming that the graph is described by a $VxV$ adjacency matrix where $V$ is the number of vertices, then the pseudocode of this algorithm below, shows how the solution is obtain in $V$ stages, where each stage involves the update of the distance matrix via the involvement of an intermediate vertex:

```
// initialization phase
Allocate a VxV distance matrix, and set all elements to infinity
for each vertex v
   distance[v][v] <- 0
for each edge (u,v)
   distance[u][v] <- adj(u,v)  // the weight of the edge (u,v)

// computation phase
for k from 1 to V   // for each k intermediate vertex
   for i from 1 to V  // reconsider each pair of vertices
      for j from 1 to V
         if distance[i][j] > distance[i][k] + distance[k][j]
             distance[i][j] <- distance[i][k] + distance[k][j]
```

```
        end if
return distance
```

Create a Thrust program that could perform the $V$ individual stages of the algorithm in parallel, i.e. parallelize the two inner loops of the algorithm.

**Answer**

Similarly to the previous exercise for finding the pair of most distant points in a set, we can use a `thrust::counting_iterator` to go over all the elements of the adjacency matrix. Fortunately, as all elements are to be considered[1], the mapping between an index and the adjacency matrix coordinates is straightforward as shown in lines 31,32.

The functor that is used to update the distance of two vertices (lines 22-41), is setup so that it has access to the adjacency matrix data and the number of vertices $V$ (lines 64,65). Before each phase of the algorithm executed by the `for` loop of lines 68-76, the index of the vertex to be used as an intermediate, is also set (line 71).

```
1   #include <thrust/host_vector.h>
2   #include <thrust/device_vector.h>
3   #include <thrust/iterator/counting_iterator.h>
4   #include <thrust/transform.h>
5   #include <thrust/random.h>
6   #include <stdio.h>
7
8   using namespace std;
9
10  //**************************************************
11  void printAdj (thrust::host_vector < int >&a, int V)
12  {
13    for (int i = 0; i < V; i++)
14      {
15        for (int j = 0; j < V; j++)
16          printf ("%3i ", a[i * V + j]);
17        printf ("\n");
18      }
19  }
20
21  //**************************************************
22  struct floydFunct
23  {
24    thrust::device_ptr < int >adj;
25    int V;
26    int k;
27
28    // comparing distances of pairs of points A and B as indexed by↩
         i,j
29    __host__ __device__ int operator () (int idx)
30    {
31      int row = idx / V;
32      int col = idx % V;
33
34      int currDist = adj[idx];
35      int otherRouteDist = adj[row * V + k] + adj[k * V + col];
36      if (otherRouteDist < currDist)
37          return otherRouteDist;
38      else
39          return currDist;
40    }
41  };
42
```

[1]Actually the $k^{th}$ column and row are unchanged during stage $k$ but we do not exclude them from the computation as this would complicate the code. Not only do we need to change the mapping from the `thrust::counting_iterator` to the table cells, but also the results of each stage's computation cannot be stored in-place.

```
//**************************************************
int main (int argc, char **argv)
{
  // initialize the RNG
  thrust::default_random_engine rng (time (0));
  thrust::uniform_int_distribution < int >uniDistr (1, 100);

  int V = atoi (argv[1]);

  // generate the data on the host and move them to the device
  thrust::device_vector < int >d_adj (V * V);
  thrust::host_vector < int >h_adj (V * V);
  for (int i = 0; i < V * V; i++)
    h_adj[i] = uniDistr (rng);
  for (int i = 0; i < V; i++)
    h_adj[i * V + i] = 0;

  d_adj = h_adj;

  // initialize the functor so that it has access to the coordinate arrays
  floydFunct f;
  f.V = V;
  f.adj = d_adj.data ();

  printAdj (h_adj, V);
  for (int k = 0; k < V; k++)
    {
      // set the functor to use the k-th node as intermediate
      f.k = k;
      thrust::transform (thrust::counting_iterator <int>(0),
                         thrust::counting_iterator <int>(V * V),
                         d_adj.begin (),
                         f);
    }

  printf ("----------------\n");
  h_adj = d_adj;
  printAdj (h_adj, V);

  return 0;
}
```

# Chapter 8

# Load Balancing

## Exercises

1. Design a brute-force/exhaustive algorithm for determining the optimum subset of $N$ nodes/processors in the case of static load balancing. What is the complexity of your algorithm?

   **Answer**

   In general, finding the optimum subset of nodes requires enumerating all the possible subsets, i.e. $2^N - 1$ if we ignore the empty set. The enumeration can be implemented in a variety of ways, the following recursive algorithm being a generic approach that does not impose any restrictions on $N$.

   The following description is based on the assumption that a function to evaluate the execution time on a subset $S$ exists (symbolic name `eval()` used to represent this function).

   The recursive auxiliary function OptimumSetAux, calls itself by including (line 15) or excluding (line 16) the $i$-th element of the processor set $P$, until the parameter $i$ becomes equal to $N$ (base case of line 8), which indicates that a subset is ready to be evaluated.

```
 1: procedure OptimumSet($P$)
 2:     $bestS \leftarrow \emptyset$
 3:     $bestT \leftarrow \inf$
 4:     OptimumSetAux($P$,$\emptyset$, 0, $bestS$, $bestT$ )
 5:     Return $bestS$
 6: end procedure
 7: procedure OptimumSetAux($P$, $S$, $i$, $bestS$, $bestT$ )
 8:     if $i = N$ then
 9:         $T \leftarrow eval(S)$
10:         if $T < bestT$ then
11:             $bestS \leftarrow S$
12:             $bestT \leftarrow T$
13:         end if
14:     else
15:         OptimumSetAux($P$, $S \cup P_i$, $i+1$, $bestS$, $bestT$ )
```

```
16:         OptimumSetAux($P$, $S$, $i+1$, $bestS$, $bestT$ )
17:     end if
18: end procedure
```

The complexity of this algorithm depends on the cost of the `eval()` function. Given that there are $2^N$ subsets to be evaluated, we can conclude that the complexity $\in \Omega(2^N)$.

2. Design a greedy algorithm for determining the optimum subset of $N$ nodes/processors in the case of static load balancing. Will you always get the same solution as the one provided by an exhaustive algorithm?

   **Answer**

   A greedy algorithm can be employed that initially sorts the nodes in descending order of their computational power, i.e. ascending of their $p_i$ parameter as identified in Table 8.1.

   The following algorithm iterates over the sorted nodes, adding one node at a time for as long as the execution time decreases (`while` loop terminates otherwise via line 14) or until all nodes are examined (line 6). The complex condition of line 9 ensures that sets leading to invalid solutions (i.e. with one or more $part_k \leq 0$) are not considered.

   Actually, the condition in line 9 can be simply "`if` $T < bestT$ `then`", as a negative part to a node enlarges the parts assigned to other nodes, inevitable making the execution time higher.

```
1: procedure GreedySet($P$)
2:     $P' \leftarrow sort(P)$
3:     $bestS \leftarrow P'_0$
4:     $bestT \leftarrow eval(bestS)$
5:     $i \leftarrow 1$
6:     while $i < N$ do
7:         $S \leftarrow bestS \cup P'_i$
8:         $T \leftarrow eval(S)$
9:         if $T < bestT$ AND $\forall P_k \in S, part_k > 0$ then
10:            $bestS \leftarrow S$
11:            $bestT \leftarrow T$
12:            $i \leftarrow i + 1$
13:        else
14:            Break
15:        end if
16:    end while
17:    Return $bestS$
18: end procedure
```

   A heuristic algorithm cannot produce an identical solution to an exhaustive algorithm, unless this is done by chance, or special circumstances/problem settings allow the generation of an optimum solution via a greedy approach.

3. The closed-form solutions in Section 8.3.3.1 for the N-port communication setup, are based on the assumption that the master participates in the processing of the load. Derive the equations that would govern the solution

if the master abstained from this task, facilitating only I/O functionality instead.

**Answer**

The answer to this question is already provided in Section 8.3.3.1, albeit in a condensed form, without a length explanation. This exercise serves the purpose of testing student understanding and their ability to follow the proper steps towards the derivation of a solution.

Once we remove the master node from computation duties, Equation 8.9 becomes:

$$\begin{aligned} t_{distr}^{(i)} + t_{comp}^{(i)} + t_{coll}^{(i)} = t_{distr}^{(j)} + t_{comp}^{(j)} + t_{coll}^{(j)} \Rightarrow \\ l_i(a\ part_i L + b) + p_i(part_i L + e_i) + l_i(c\ part_i L + d) = \\ l_j(a\ part_j L + b) + p_j(part_j L + e_j) + l_j(c\ part_j L + d) \end{aligned} \tag{8.1}$$

To solve the problem, we need to find one of the assigned parts, e.g. $part_1$. So from the above equation we can express all $part_i$ with $i \neq 1$ as a function of $part_1$:

$$part_i = part_1 \frac{l_1(a+c)+p_1}{l_i(a+c)+p_i} + \frac{p_1 e_1 - p_i e_i + (l_1 - l_i)(b+d)}{L(l_i(a+c)+p_i)} \tag{8.2}$$

Then, the normalization equation allows us to get a solution for $part_1$:

$$\begin{aligned} &\sum_{j=1}^{N-1} part_j = 1 \Rightarrow \\ &part_1 + part_1 \frac{l_1(a+c)+p_1}{l_2(a+c)+p_2} + \frac{p_1 e_1 - p_2 e_2 + (l_1 - l_2)(b+d)}{L(l_2(a+c)+p_2)} + \dots \\ &+ part_1 \frac{l_1(a+c)+p_1}{l_{N-1}(a+c)+p_{N-1}} + \frac{p_1 e_1 - p_{N-1} e_{N-1} + (l_1 - l_{N-1})(b+d)}{L(l_{N-1}(a+c)+p_{N-1})} = 1 \Rightarrow \\ &part_1 \left(1 + \sum_{j=2}^{N-1} \frac{l_1(a+c)+p_1}{l_j(a+c)+p_j}\right) = 1 + \sum_{j=2}^{N-1} \frac{p_j e_j - p_1 e_1 + (l_j - l_1)(b+d)}{L(l_j(a+c)+p_j)} \Rightarrow \\ &part_1 \sum_{j=1}^{N-1} \frac{l_1(a+c)+p_1}{l_j(a+c)+p_j} = 1 + \sum_{j=2}^{N-1} \frac{p_j e_j - p_1 e_1 + (l_j - l_1)(b+d)}{L(l_j(a+c)+p_j)} \Rightarrow \\ &part_1 = \frac{1 + L^{-1} \sum_{j=2}^{N-1} \frac{p_j e_j - p_1 e_1 + (l_j - l_1)(b+d)}{l_j(a+c)+p_j}}{(l_1(a+c)+p_1) \sum_{j=1}^{N-1} (l_j(a+c)+p_j)^{-1}} \end{aligned} \tag{8.3}$$

4. Write a program that calculates $part_0$ from Equations 8.11 and 8.39 in a linear number of steps.

**Answer**

The following listing contains two functions that calculate $part_0$ and subsequently all $part_i$, for the two problem settings that Equations 8.11 and 8.39 correspond to. The “trick” (especially for Equation 8.39 that involves double summations) is to pre-compute and reuse terms, such as e.g. the

$\sum_{j=1}^{k} b_j$ term of Equation. 8.39, via lines 51-53 in `OneportBlockSingleInst`. The two functions also return the predicted total execution time.

It is important to note that `OneportBlockSingleInst` just implements Equations 8.38 and 8.39, without ordering the nodes as mandated by Theorem 8.3.3.

```
 1  #include <iostream>
 2
 3  using namespace std;
 4  //**********************************************
 5  // Stores the computed part_i in array part and the returns the ↩
        total execution time
 6  double NportBlockSingleInst (double *p, double *e, double *l, ↩
        double a, double b, double c, double d, int N, long L, ↩
        double *part)
 7  {
 8    double lacp[N];  // should be replaced by dynamic allocation if↩
          N is big
 9    double sumTerm[N];
10    double totalTime;
11    double nomin, denom;
12
13    for (int i = 0; i < N; i++)
14      lacp[i] = l[i] * (a + c) + p[i];
15
16    for (int i = 1; i < N; i++)
17      sumTerm[i] = p[0] * e[0] - p[i] * e[i] - l[i] * (b + d);
18
19    nomin = 0;
20    for (int i = 1; i < N; i++)
21      nomin += sumTerm[i] / lacp[i];
22    nomin = 1 - nomin / L;
23
24    denom = 1;
25    for (int i = 1; i < N; i++)
26      denom += 1 / lacp[i];
27    denom *= p[0];
28
29    part[0] = nomin / denom;
30
31    for (int i = 1; i < N; i++)
32      part[i] = part[0] * p[0] / lacp[i] + sumTerm[i] / (L * lacp[i↩
            ]);
33
34    totalTime = p[0] * (part[0] * L + e[0]);
35    return totalTime;
36  }
37
38  //**********************************************
39  // Stores the computed part_i in array part and the returns the ↩
        total execution time
40  double OneportBlockSingleInst (double *p, double *e, double l, ↩
        double *b, int N, long L, double *part)
41  {
42    double pl[N]; // should be replaced by dynamic allocation if N ↩
         is big
43    double bSum[N];
44
45    double totalTime;
46    double nomin, denom;
47
48    for (int i = 0; i < N; i++)
49      pl[i] = p[i] + l;
50
51    bSum[0] = 0;
52    for (int i = 1; i < N; i++)
53      bSum[i] = b[i] + bSum[i - 1];
54
55    nomin = 0;
56    for (int i = 1; i < N; i++)
57      nomin += l / pl[i] * bSum[i];
58    nomin /= L;
```

```
  nomin++;
  for (int i = 1; i < N; i++)
    nomin += 1 / pl[i];

  denom = 0;
  for (int i = 0; i < N; i++)
    denom += 1 / pl[i];
  denom *= pl[0];

  part[0] = nomin / denom;

  for (int i = 1; i < N; i++)
    part[i] = part[0] * pl[0] / pl[i] - 1 / (L * pl[i]) * bSum[i]↩
         - 1 / pl[i];

  totalTime = p[0] * part[0] * L;
  return totalTime;
}

//*********************************************
int main ()
{
  //small tester data
  double p[4] = { 1, 2, 3, 4 };
  double e[4] = { 1, 1, 1, 1 };
  double b[4] = { 1000, 1000, 1000, 1000 };
  double l[4] = { 1, 1, 1, 1 };
  double part[4];

  cout << "t : " << NportBlockSingleInst (p, e, 1, 1, 0, 1, 0, 4,↩
       1000000, part) << endl;
  for (int i = 0; i < 4; i++)
    cout << part[i] << " ";
  cout << endl;

  cout << "t : " << OneportBlockSingleInst (p, e, 1, b, 4, ↩
      1000000, part) << endl;
  for (int i = 0; i < 4; i++)
    cout << part[i] << " ";
  cout << endl;

  return 0;
}
```

5. Solve the example at the end of Section 8.3.3.2, by reversing the computing power of the nodes: $p_0 = 4 \cdot 1.631 \cdot 10^{-07}$, $p_1 = 3 \cdot (1.631 \cdot 10^{-07})$, $p_2 = 2 \cdot (1.631 \cdot 10^{-07})$ and $p_3 = (1.631 \cdot 10^{-07})$.

   **Answer**

   Before applying the equations, the nodes (excluding the master) should be sorted according to their $b_i(p_i + l)$ property. This presents a challenge, as while we can switch the place of a node, the $b_i$ overheads are node *order* specific and not node specific. Hence. changing the order of a node, potentially causes a change in its $e_i$ parameter. Fortunately, the closed-form solutions of Section 8.3.3.2 allow the easy calculation of the predicted time for the different permutations of the nodes (surely not advisable for large $N$, but quite doable for the particular problem).

   The results show that load should be distributed in the order $P_3$, $P_2$, $P_1$. The corresponding parts are $part_0 = 0.13516$, $part_3 = 0.465104$, $part_2 = 0.238957$ and $part_1 = 0.160779$, with a total execution time of 2.19sec.

6. An alternative to having the master node distribute the load, is to have all the nodes access a network filesystem and retrieve the data from there.

What kind of communication configuration would correspond to such an arrangement? Calculate the partitioning that would be produced in this case, for the same problem setting as the previous question.

**Answer**

Such a design corresponds to a N-port communication configuration. If we assume that all nodes (including $P_0$ which was previously considered as having the load locally) perform the same sequence of getting the load, processing it and sending the results back to the filesystem server, then the minimum execution time is achieved by having all nodes finish at the same time instance, i.e. for every pair of $P_i$ and $P_j$, we have:

$$\begin{aligned}
&t_{distr}^{(i)} + t_{comp}^{(i)} + t_{coll}^{(i)} = t_{distr}^{(j)} + t_{comp}^{(j)} + t_{coll}^{(j)} \Rightarrow \\
&l(part_i L + b_i) + p_i\, part_i L + l\, part_i L = l(part_j L + b_j) + p_j\, part_j L + l\, part_j L \Rightarrow \\
&part_i L(p_i + 2\ l) = part_j L(p_j + 2\ l) + l(b_j - b_i) \Rightarrow \\
&part_i = part_j \frac{p_j + 2\ l}{p_i + 2\ l} + \frac{l(b_j - b_i)}{L(p_i + 2\ l)} \qquad (8.4)
\end{aligned}$$

Thus, we can express all $part_i$ as a function of $part_0$:

$$part_i = part_0 \frac{p_0 + 2\ l}{p_i + 2\ l} + \frac{l(b_0 - b_i)}{L(p_i + 2\ l)} \qquad (8.5)$$

The normalization equation can then yield a closed-form solution for $part_0$:

$$\begin{aligned}
&\sum_{i=0}^{N-1} part_i = 1 \Rightarrow \\
&part_0 \sum_{i=0}^{N-1} \frac{p_0 + 2\ l}{p_i + 2\ l} + \frac{l}{L} \sum_{i=1}^{N-1} \frac{b_0 - b_i}{p_i + 2\ l} = 1 \Rightarrow \\
&part_0 = \frac{1 + \frac{l}{L} \sum_{i=1}^{N-1} \frac{b_i - b_0}{p_i + 2\ l}}{(p_0 + 2\ l) \sum_{i=0}^{N-1} (p_i + 2\ l)^{-1}} \qquad (8.6)
\end{aligned}$$

If we assume that the nodes are ordered as specified by the problem statement, then we have $b_1 = b_2 = 2 \cdot 3 \cdot 2160B$ and $b_0 = b_3 = 3 \cdot 2160B$. Using the above equations produces $part_0 = 0.125605$, $part_1 = 0.165952$, $part_2 = 0.24447$ and $part_1 = 0.463997$, with a total execution time of 2.097sec.

7. The partitioning performed using DLT in the kernel convolution example, returns the percent of the input bytes that should be assigned to each node? How can we convert this number to pixels? One may argue that we should instead use image rows. How can we convert the result of our analysis to rows?

   **Answer**

   As long as there is a linear relationship between the data volume and the desired load unit, there is no need for a conversion. The $part_i$ results are percentages and can be used without conversion to split the number of rows as well as the number of pixels.

8. Use the `tiobench` utility available at http://sourceforge.net/projects/tiobench/ to measure the performance you can get by concurrent access to a network filesystem (e.g. a NFS volume). Calculate the collective throughput of the server versus the number of threads used.

   **Answer**

   No model answer is available for this exercise. The students should experiment with the `tiobench` utility and plot the results. A plot similar to Figure 8.6 should be produced.

9. One way that can be used to improve the distribution cost of input data is to compress them. What kind of compression algorithms could be used in this case? How could we adapt the cost models to reflect this change?

   **Answer**

   There are two distinct possibilities:

   (a) The input data are already in compressed form. This imposed restrictions on the compression algorithm and data encoding format, as they should permit random access to the input data. This immediately rules out dynamic compression schemes (e.g. dictionary based) and enforces some sort of partitioning of the compressed data, in the form of packets or blocks. A static Huffman/arithmetic code compression algorithm would be suitable for the task.

   (b) The data are compressed before the distribution phase. Any compression algorithm could be used in this case, as long as the compression-decompression overhead does not exceed the gains from the reduced communication time.

   Let's assume that a node decompresses the input upon receiving it, and compresses the results prior to sending them back, spending $compr_i$ time per load unit. If on average, the compression ratio is $cr = \frac{originalDataSize}{compressedDataSize}$, then, the cost models would have to be modified as follows:

$$t_{comp}^{(i)} = p_i(part_i L + e_i) + compr_i part_i L = (p_i + compr_i)(part_i L + \frac{p_i e_i}{p_i + compr_i}) \quad (8.7)$$

$$t_{distr}^{(i)} = l_i(a\ part_i \frac{L}{cr} + b) = l_i(\frac{a}{cr} part_i L + b) \quad (8.8)$$

$$t_{coll}^{(i)} = l_i(c\ part_i \frac{L}{cr} + d) = l_i(\frac{c}{cr} part_i L + d) \quad (8.9)$$

   Equations 8.7-8.9 adhere to the formulation of the book's Equations 8.5, 8.7 and 8.8(see pages 583, 585 and 586). This is a clear indication that existing analysis results could be applicable for such a design, by adapting the model parameters.

10. Use the DLTlib library to calculate the optimum partitioning for a "query processing" application, where the communication cost is independent of the workload, consisting of a query during distribution and the result during the collection. You can assume that $b = d = 1$, the workload consists of $L = 10^6$ items, and the parallel platform is made of 10 compute nodes connected in a single level tree, with $p_i = (i+1) \cdot 0.01 sec/item \ \ \forall i \in [0, 9]$ and $l = 0.001 sec/item$.

    What would happen if the communication was 10 times slower?

    **Answer** The following listing illustrates how the appropriate DLTlib function can be called and the results printed-out:

```
#include <time.h>
#include <stdio.h>
#include <iostream>

using namespace std;

//-----------------------------------------------------------------
// DLTlib specific definitions that need to be used by the ↩
    library.
long global_random_seed;
#define MAX_NODE_DEGREE 10
#include "dltlib.cpp"
//-----------------------------------------------------------------

int main ()
{
  double p = 0.01;
  double l = 0.001;
  long int L = 1000000;
  double b = 1, d = 1;
  // STEP 1
  Network platform;            // object representing parallel ↩
      platform

  // STEP 2
  // insert one-by-one the nodes that make up the machine. P0 by ↩
      default participates in
  // the computation
  platform.InsertNode ((char *) "P0", p, 0, (char *) NULL, l, ↩
      true);
  for (int i = 1; i < 10; i++)
    {
      char buff[10];
      sprintf (buff, "P%i", i);
      platform.InsertNode (buff, (i + 1) * p, 0, (char *) "P0", l↩
          , true);
    }

  // STEP 3
  // Solve the partitioning problem for 1-port, block-type ↩
      computation
  platform.SolveQuery (L, b, d);

  // print out the results, if the solution is valid
  if (platform.valid == 1)
    {
      cout << "Predicted execution time: " << platform.↩
          SimulateQuery (b, d, 0) << endl;

      // STEP 4
      // Compute nodes are stored in a public linked-list that ↩
          allows
      // rearrangement to the order of distribution and ↩
          collection
      cout << "Solution in terms of load percent :\n";
      Node *h = platform.head;
      while (h != NULL)
        {
            // Single installment case...
            cout << h->name << "\t" << h->part << endl;
            h = h->next_n;
          }
      }
    else
      cout << "Solution could not be found\n";
    return 0;
}
```

A noteworthy addition to the sample provided in the book, is line 10 that changes the maximum number of child nodes of a machine. The default of the library is only four, far less than the problem requirement of nine, which if left unchanged would cause the program to fail during execution with a protection fault.

This program can be compiled and run as follows:

```
$ g++ DLTexercise.cpp -o DLTexercise -I ../../DLTlib -lglpk
$ ./DLTexercise
Predicted execution time: 3414.18
Solution in terms of load percent :
P0      0.341418
P1      0.170709
P2      0.113806
P3      0.0853543
P4      0.0682834
P5      0.0569028
P6      0.0487738
P7      0.042677
P8      0.0379351
P9      0.0341416
```

Finally, testing for a 10 times slower communication medium is as simple as changing the `l` variable value of line 17 to `0.01`. The reported results are almost identical to the previous ones, as the communication cost is negligible in comparison to the computation:

```
Predicted execution time: 3414.22
Solution in terms of load percent :
P0      0.341422
P1      0.17071
P2      0.113806
P3      0.085354
P4      0.0682828
P5      0.056902
P6      0.0487729
P7      0.042676
P8      0.037934
P9      0.0341404
```

11. Derive the equivalent of Equations 8.47 and 8.48 for arbitrary $a$ and $c$, and implement them as part of a partitioning function.

**Answer**

The question is essentially to solve the N-port, block-type, single-installment distribution problem (i.e. what Section 8.3.3.1 covers) for the case when $P_0$ is not the load originating node. As such, $P_0$ has to incur communication cost both for the distribution and the result collection phases.

The solution is identical to the one in Exercise 3. The only difference is that while in Exercise 3 we assume that $P_0$ is not participating in the load processing, in this exercise we assume that $P_0$ does not hold the load originally, i.e. there is another master node that is only distributing the data. We can reuse the solution of Exercise 3, with a simple change of the indices, so that all $part_i$ are expressed as functions of $part_0$, i.e.:

$$part_i = part_0 \frac{l_0(a+c)+p_0}{l_i(a+c)+p_i} + \frac{p_0 e_0 - p_i e_i + (l_0 - l_i)(b+d)}{L(l_i(a+c)+p_i)} \tag{8.10}$$

and

$$part_0 = \frac{1 + L^{-1}\sum_{j=1}^{N-1} \frac{p_j e_j - p_0 e_0 + (l_j - l_0)(b+d)}{l_j(a+c)+p_j}}{(l_0(a+c)+p_0)\sum_{j=0}^{N-1}(l_j(a+c)+p_j)^{-1}} \tag{8.11}$$

The following C++ code implements these two Equations, with function `NportBlockSingleInst` returning the $part_i$ and total execution time:

```cpp
#include <iostream>

using namespace std;
//**********************************************
double NportBlockSingleInst (double *p, double *e, double *l, ↩
    double a, double b, double c, double d, int N, long L, ↩
    double *part)
{
  double lacp[N];  // should be replaced by dynamic allocation if↩
       N is big
  double sumTerm[N];
  double totalTime;
  double nomin, denom;

  for (int i = 0; i < N; i++)
    lacp[i] = l[i] * (a + c) + p[i];

  for (int i = 1; i < N; i++)
    sumTerm[i] = p[i] * e[i] - p[0] * e[0] + (l[i] - l[0]) * (b +↩
         d);

  nomin = 0;
  for (int i = 1; i < N; i++)
    nomin += sumTerm[i] / lacp[i];
  nomin = 1 + nomin / L;

  denom = 0;
  for (int i = 0; i < N; i++)
    denom += (1 / lacp[i]);
  denom *= lacp[0];

  part[0] = nomin / denom;

  for (int i = 1; i < N; i++)
    part[i] = part[0] * lacp[0] / lacp[i] - sumTerm[i] / (L * ↩
        lacp[i]);

  totalTime = p[0] * (part[0] * L + e[0]) + l[0] * (a * part[0] *↩
       L + b) + l[0] * (c * part[0] * L + d);
  return totalTime;
}


//**********************************************
int main ()
{
  double p[4] = { 1, 2, 3, 4 };
  double e[4] = { 1, 1, 1, 1 };
  double b[4] = { 1000, 1000, 1000, 1000 };
  double l[4] = { 1, 1, 1, 1 };
  double part[4];

  cout << "t : " << NportBlockSingleInst (p, e, l, 1, 0, 1, 0, 4,↩
       1000000, part) << endl;
  for (int i = 0; i < 4; i++)
```

```
    cout << part[i] << " ";
  cout << endl;

  return 0;
}
```

12. The DLT examples presented in Sections 8.3.3.1 and 8.3.3.2, do not address an aspect of the problem which is significant : what is the optimum subset of nodes to use to process a load? Implement the heuristic algorithm you came up with in Exercise 2, to derive such a set.

    Hint: nodes that should not be part of this set, get a negative load assignment $part_i$.

    **Answer**

    The effects of node choice and order are more significant when a one-port configuration is used. When N-port configurations are used, one needs to deal with the node selection problem, only when the communication cost to a node can exceed the total execution time on the other nodes.

    In the following listing we implement the heuristic outlined in Exercise 2. The program tests for a maximum number of nodes, as specified in the command line, and reports the optimum number of nodes to be used and associated partitioning. The nodes are sorted (actually they are generated in this fashion) in descending order of their computing power.

```
#include <iostream>
#include <stdlib.h>

using namespace std;

//***********************************************
double OneportBlockSingleInst (double *p, double *e, double l, ↩
    double *b, int N, long L, double *part)
{
  double pl[N]; // should be replaced by dynamic allocation if N ↩
      is big
  double bSum[N];

  double totalTime;
  double nomin, denom;

  for (int i = 0; i < N; i++)
    pl[i] = p[i] + l;

  bSum[0] = 0;
  for (int i = 1; i < N; i++)
    bSum[i] = b[i] + bSum[i - 1];

  nomin = 0;
  for (int i = 1; i < N; i++)
    nomin += l / pl[i] * bSum[i];
  nomin /= L;
  nomin++;
  for (int i = 1; i < N; i++)
    nomin += l / pl[i];


  denom = 0;
  for (int i = 0; i < N; i++)
    denom += l / pl[i];
  denom *= pl[0];

  part[0] = nomin / denom;

  for (int i = 1; i < N; i++)
```

```
    part[i] = part[0] * pl[0] / pl[i] - l / (L * pl[i]) * bSum[i] - l / pl[i];

  totalTime = p[0] * part[0] * L;
  return totalTime;
}

//**********************************************
int main (int argc, char **argv)
{
  int N = atoi (argv[1]);

  double p[N];
  double e[N];
  double b[N];
  double l = .1;
  double part[N];
  long L = 100;

  for (int i = 0; i < N; i++)
    {
      p[i] = (i + 1) * 1;
      e[i] = 0.1;
      b[i] = 1;
    }

  double bestT = p[0] * L + e[0], bestN = 1;
  for (int i = 1; i < N; i++)
    {
      double t = OneportBlockSingleInst (p, e, 1, b, i + 1, L, part);
      bool valid = true;
      for (int j = 0; j < i + 1 && valid; j++)
        if (part[j] < 0)
          valid = false;
      cout << i + 1 << " " << t << " " << valid << endl;     // debugging output
      if (t < bestT)  // valid does not need to be checked as per the comments in Exercise 2
        {
          bestN = i + 1;
          bestT = t;
        }
    }

  cout << "Best time " << bestT << ", best N " << bestN << endl;
  OneportBlockSingleInst (p, e, 1, b, bestN, L, part);  // solve again to obtain part_i
  for (int i = 0; i < bestN; i++)
    cout << part[i] << " ";
  cout << endl;

  return 0;
}
```

A sample of the generated output is shown below:

```
$ ./heuristicsDLT 50
2 80.2 1
3 73.4615 1
4 70.039 1
5 67.9655 1
6 66.5785 1
7 65.5904 1
8 64.8559 1
9 64.2931 1
10 63.8524 1
...
23 62.1486 1
24 62.1394 1
25 62.1375 1
26 62.1422 0
27 62.1526 0
28 62.1683 0
```

```
...
50 63.2093 0
Best time 62.1375, best N 25
0.621375 0.0775835 0.0556876 0.0425501 0.0337917 0.0275358 ↩
    0.0228438 0.0191945 0.016275 0.0138864 0.0118959 0.0102116 ↩
    0.00876789 0.0075167 0.0064219 0.00545591 0.00459725 ↩
    0.00382897 0.00313752 0.00251193 0.0019432 0.00142393 ↩
    0.000947934 0.000510017 0.000105786
```