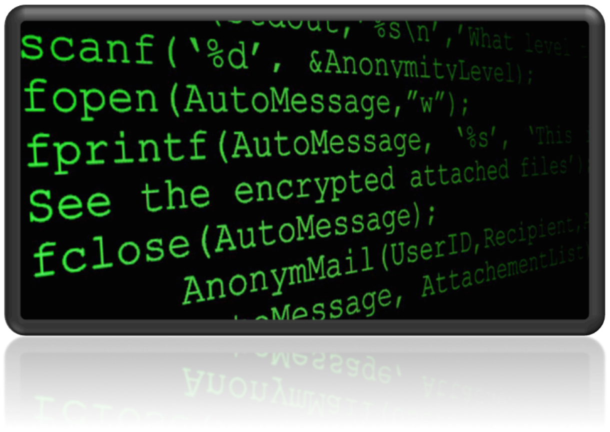
Comp 428/6281
Parallel Computing

Final Project



```
scanf('%d', &AnonymityLevel);
fopen(AutoMessage,"w");
fprintf(AutoMessage, '%s', 'This
See the encrypted attached files')
fclose(AutoMessage);
     AnonymMail(UserID,Recipient,
     Message, AttachementList
```

Due date: 11:55 PM, Monday, December 7.

Note: All submissions are made via Moodle (see "deliverables" section). Late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied automatically.

After you upload the project, please verify that your submission is recorded as submitted.

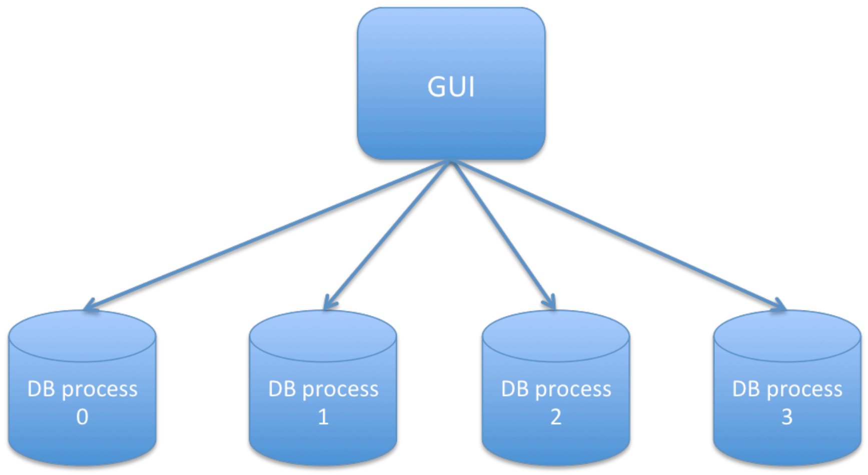
Note: Graduate students will be required to solve additional problems and/or provide additional software components. These additional components will be clearly indicated in the description below.

Total Points for Undergads: 40
Total Points for Grad Students: 50

Overview. The project will allow you to build on some of the concepts and skills that you have developed throughout the course. If you have done the assignments, many of requirements of the project should at least seem familiar.

To begin, your basic objective will be to build a small data management application. In particular, you will provide a relatively simple, in-memory DBMS that will support interactive queries on a Sales database. Note that by "simple", we really do mean simple. A relational DBMS is an extraordinarily complex piece of software, so our goal is just to support a handful of basic queries.

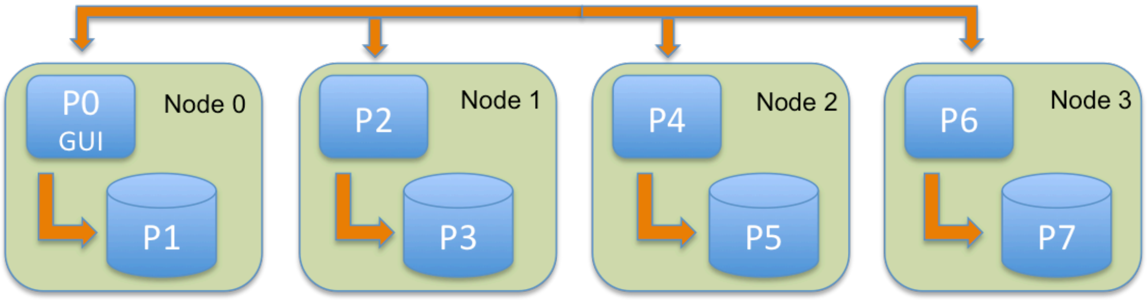
Of course, you have to do this in parallel. So the idea then is to provide a simple GUI through which the user can make a couple of basic but hopefully useful queries. Conceptually, this DBMS would look like the following:



Details. Okay, so that sounds easy enough. As the saying goes, however, the "devil is in the details". For example, we need data for our database. Like any realistic environment, the data should be inserted over time. It might not be immediately obvious, but we also need to be able to process new queries while data is being loaded. We want to make sure that the system is responsive so that there are no blocking operations that would prevent queries from being executed efficiently - this would defeat the purpose of using a parallel system. And, of course, we must be able to actually answer the queries correctly and provide output in a meaningful way.

So our simple model must be extended. In particular, each physical node in the system will actually house two MPI processes. One will be an *even numbered* process (P_{even}) and one will be *odd numbered* (P_{odd}). For example, Node 0 will have P0 and P1, while Node 3 will have P6 and P7. The P_{even} process will be responsible for accepting the original query and preparing the final results. In turn the P_{odd} process will represent the DBMS itself. In other words, its job is to store and retrieve the raw data that P_{even} will use to complete the query request.

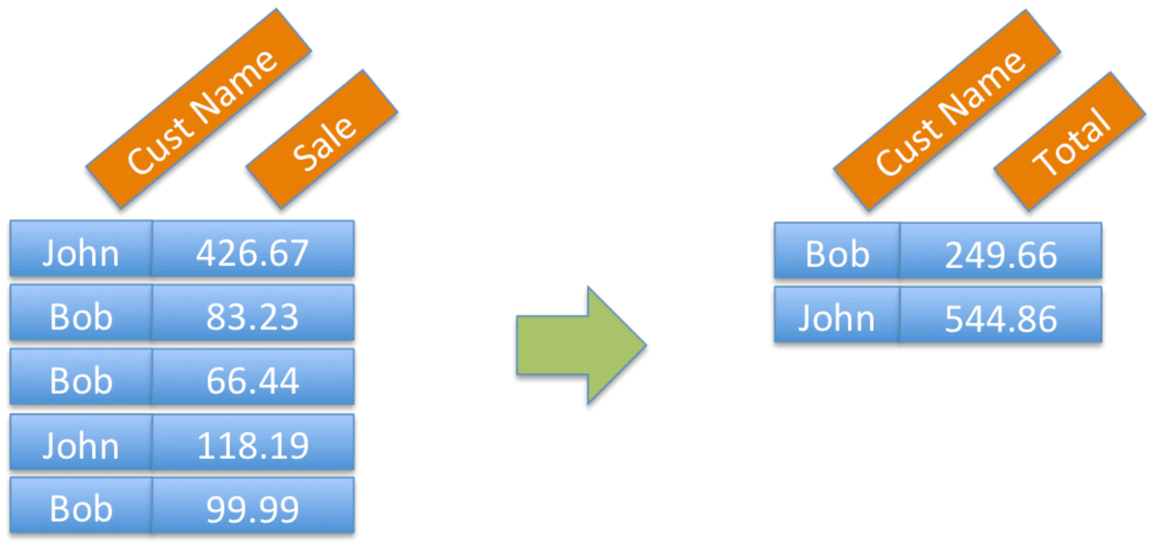
If this is not clear, the following diagram shows the detailed model.



So let's follow the process from end to end. When the program starts, process P0 will display a very simple menu that allows the user to define a query (more on this below). This menu will be very similar to the one developed in A1. Once the query is defined, P0 will send the query to all even-numbered processes (including itself).

The even numbered processes receive the query. They then pass the query to their odd-numbered partner, who will retrieve the matching records from its *in-memory* table of records (the data schema will be defined soon). The odd numbered process returns the data to its partner (remember, both are on the same physical node). Each even numbered process now has a portion of the result. These results must be merged together in order to get a final result.

For the "merge", you will use a simple parallel bucket sort amongst the even numbered processes (there is no data skew so a true Sample Sort is not required). The "sort key" for the Buckets will depend on the specific query. In any case, once the data arrives on the target nodes/buckets, final query processing is done. In this case, all that is required is for the matching records to be summed. See the example below for an example using customer sales records.



To provide the result to the user, the results are now collected at Process 0, where they will de displayed to the console. When the user hits "Press key to continue", the user will be prompted for a new query (or exit).

Odd Numbered Processes. The job of these processes is to hold the data in memory and provide it as required. The data itself will simply be stored in a single table, which is really just a big array.

So where do the data records come from? Each node will be associated with a text file called data#.txt. (for # = 0 to 4). These are human readable text files that are produced by a data generator application. You do not have to create the files yourself – samples are included with the project. The files will be relatively large – the samples have 50000 lines per file, but the final grading files may be considerably larger. The idea is that these text files will simulate incoming data streams. Basically, the idea is to read some of the data at random time intervals (between 2 and 5 seconds between each read).

The actual record count per read will be determined by a command line parameter called *read_max*. This parameter will be a value between 1000 and 100000. Basically, you will generate a number between 10 and *read_max*. You will then read this many records from the local file. Each process must of course generate unique random numbers.

As a concrete example, if *read_max* was 1000, we might average 500 records every 3 seconds. In this scenario, your 50K record sample files would allow the application to run for about 5 minutes before it ran out of data.

The random addition of data to the in-memory DB is not hard in itself. You will simply read the records from the text file line by line, splitting on the field delimiters, and appending them to the table. The real problem is that the process can't wait for a query from its partner (via a blocking MPI receive) and wait for the next data input request at the same time.

So we need to be clever. To begin, we need some way to be informed that the next cycle of data input is ready. For this we will use a Unix/Linux signal handler. If you are not familiar with signal handlers, don't worry – a signal-based timer example program is provided with the project. In short, operating systems provide a timer mechanism that allows the programmer to set a "count down" timer that will expire at some point in the future (2 to 5 seconds in our case). When it does, the operating system delivers an asynchronous "signal" to the process. To utilize this technique, you must provide a signal handler function that waits for the signal. When it arrives, the OS interrupts the normal process execution and executes the signal handler function. In our case, the signal handler will do just one thing - it will set a "read pending" variable to true.

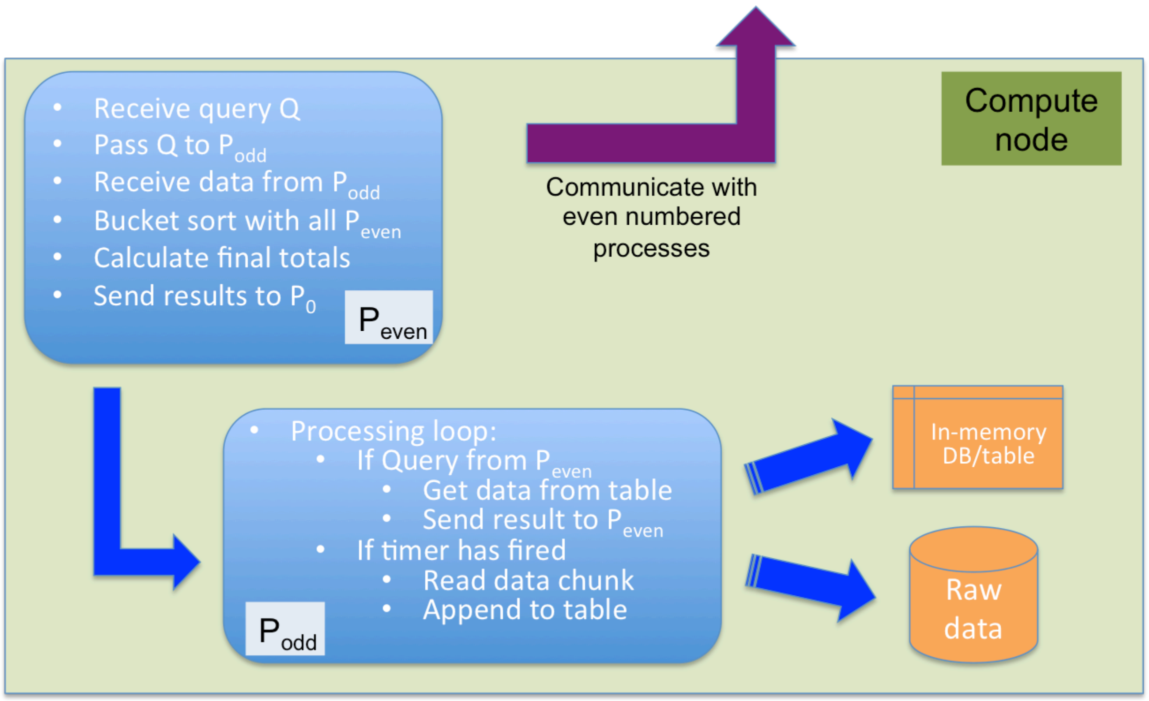
The logic for the odd numbered processes is therefore quite simple. It operates in a simple loop. It will first execute a non-blocking MPI receive to wait for the next query from P_{even}. It will then execute a very tight *polling* loop in which it checks for either of two conditions to be true:

1. The MPI receive has occurred. If so, it will retrieve the data from the table and pass it back to P_{even} .
2. The *read pending* variable has been set. If so, it will read a random number of records from the file and append it to the input array.

This loop will continue until the program terminates.

Even Numbered Processes. The job of the even numbered processes is to continuously accept query requests from P0 and provide the appropriate results. The communication itself is relatively simple. Because it only processes one query at a time, it can do a simple blocking read and wait for the next query. It communicates with P_{odd} in order to get its portion of the data. It then performs a bucket sort in order to distribute the data. Finally, it computes the summarized query result and returns the result to P0. It repeats this process until the user selects the *exit* option in the menu.

The diagram below summarizes the application logic associated with each process, both P_{even} and P_{odd}.



P0 Menu. The root process acts much like any even numbered process. It communicates with P1 to get its portion of the data and it prepares its portion of the final results. However, it must also provide the GUI and display the final output. However, this doesn't really complicate the logic much. It simple runs the GUI before it starts executing the query, and displays results after the query is processed. Otherwise, it's just like any of the other even numbered processes.

The Menu. P0 provides the query interface for the end user. This is a simple text menu, just like the first assignment. The user will be presented with a main menu that provides the basic query options:

```
Parallel DBMS Main Menu

   1. Company Sales
   2. Sales by Date
   3. Exit
```

If you select "Company Sales", the query can be executed directly. The results will be display as follows:

```
Company                  Total

APX Industries           45345.93
Nero Corp                12045.76
..
Parna Dev                34336.23
```

If you select the "Sales by Date" option, then you will be asked to provide a *date range*. The dates themselves must be provided in three parts – year, month, and day. So, for example, the Start Date would be something like:

```
Enter Starting Year:
Enter Starting Month:
Enter Starting Day:
```

When the results are printed, it would look like the following:

```
YMD               Total

2013/02/01        3455.67
2013/02/02        2605.78
..
2013/03/31        312.34
```

If the "exit" option is selected, then the application will end with a "goodbye" message.

Grad version. Graduate students will of course follow the same application logic. In addition, they will extend the system in two ways.

1] Grad students will add an additional query. Specifically, they will provide a delete query. This will be Query 3. The idea here is to use the GUI to provide a Start Date and End date combination (as with Query 2). This information will then be used to delete the relevant data on each node. In order for this to work, the P_{even} and P_{odd} nodes will have to be able to distinguish between the two types of queries.

When the query runs, P_{odd} will inform P_{even} of the number of deleted records on the local node. This information will be summed across processes and returned to P0 for display:

```
427 records were deleted from the database
```

2] You may have noticed that the communication between P_{even} and P_{odd} is not terribly efficient if the query returns a lot of data. Specifically, all those records are copied between the two processes, even though both processes are on the same physical node. In a multi-threaded application, we could share this data directly to avoid the copying. However, our MPI processes cannot do this since they really are separate processes to the operating system. We will now address this issue.

To do so, you will use the Linux IPC (Inter-process Communication) mechanism. IPC provides the ability to create a memory buffer that can be shared between distinct processes. This will allow P_{odd} to place records into the shared buffer and then have P_{even} read them from the buffer. In fact, P_{even} can do its MPI send directly from this buffer.

Unlike the signaling mechanism presented above, however, I will not be giving the grad students the source code to do the shared buffer exchange. (This is why we call you graduate students – you have to figure this out on your own). That being said, the functions used for shared memory IPC are well documented and there are many examples on the web. Feel free to use any of this. Just as a "heads up", you not only have to provide the shared buffer code itself, but you have to coordinate the exchange so that P_{even} knows when P_{odd} has actually written the data. You also do not know how many records will be returned for a given query, so the buffer should be created and then destroyed after each distinct query.

Data Schema: The schema for the DB files is as follows:

1. **sales_id**: auto-incrementing integer

2. **date:** yyyy/mm/dd format

3. **company_id**: auto-incrementing integer (starting from 1)

4. **company_name**: a set of unique strings. The count will match that of the company_id field. For example, the names in the sample data included with the project are:
    1) APX Industries
    2) Nero Corp
    3) Davidson Inc
    4) Americo
    5) Miracle Plastics
    6) Carson Engineering
    7) FormaCorp
    8) Zetta Marsten
    9) National PLC
    10) YTK Pharma
    11) Smith Jones and Davis
    12) Kelly Inc
    13) Plaster Rock PLC
    14) Bulea Ltd
    15) Yellow Co
    16) Giant Box Ltd
    17) ParaGuild Design
    18) Holistic Medicines
    19) Dyna Explosives
    20) Parna Dev

5. **sales_total** : Dollar amounts, with two digits past the decimal. The numbers are in the range 0.99 to 999.99

All records are recorded one per line, and use "|" as the field delimiter symbol. sales_id values are unique across all processes. The sample data consists of 50000 records per file, for a total of 250000 records.

A small sample is given below:

```
0|2014/10/13|10|YTK Pharma|420.31

1|2014/9/24|19|Dyna Explosives|784.75

2|2014/4/11|5|Miracle Plastics|267.89

3|2013/10/7|12|Kelly Inc|718.46
```

```
4|2014/5/11|6|Carson Engineering|810.56

5|2014/5/1|17|ParaGuild Design|843.85

6|2014/10/1|2|Nero Corp|819.22

7|2013/12/18|17|ParaGuild Design|23.57

8|2013/7/17|20|Parna Dev|838.93

9|2013/7/7|8|Zetta Marsten|236.56

10|2013/9/29|2|Nero Corp|44.21
```

Things to keep in mind. Don't forget the following:

1] You will need to ensure that there are two processes on every node. For this you must use the **npernode** option to mpirun. Basically, you must ensure (from the command line or Eclipse Run Configuration) that you have something like:

```
mpirun   -np 5   -npernode 2   my_app  1000
```

Note that the "1000" argument refers to the *read_max*.

2] You must use a second *communicator* to move data between the even numbered nodes.

3] When the user selects the "exit" option from the main menu, you cannot simply abort the whole process or just kill process 0. The exit message must be propagated to all processes (even and odd) so that they can exit properly.

4] You should collective operations whenever possible. The communication between the odd and even numbered pairs is, of course, point to point.

5] You should use derived data types when answering queries.

6] You cannot hard code any parameters related to the data. The sample data sets provided with the project are for development work. Because the final grading will be done one project at a time, we can use much larger files. There may be 5 to 10 million records in total. Moreover, there may be many more than 20 companies (with different names).

Design advice: You are free to write your code anyway that you like. However, there are a number of things that can go wrong with your code and the resulting bugs may be difficult to identify if you are trying to do too many things at once. I would suggest the following:

1] Try to set up a local Virtual Machine (VM) if at all possible. With 40 students writing long running parallel applications, each with many opportunities for bugs and deadlocks, you may find it difficult to do development work on apini in the last week of the term. Using your own VM prevents other students from interfering with your progress.

2] If using a VM, install Valgrind (instructions will be available on the web site). This will allow you identify *seg faults* and memory issues quickly.

3] Do NOT write all parts of the project at once. Create a "skeleton" program first and gradually add to it. You might so something like this:

a) Create a "skeleton" program that does nothing more than verify that you have the even/odd processes configured in the proper way.
b) Add a simple GUI to send a trivial message though the system and display the results.
c) Add the signal handler and the polling loop. At this point, however, do not read the data from disk. Just add a couple of hard coded records to the table.
d) Add the bucket sort logic. Display the raw results, but without the summarization.
e) Use Valgrind at every stage to make sure that you don't have memory management problems that will eventually corrupt your code.

At this point, you have the core logic in place. You can now add the other features – reading from the input file, data summarization, full query reports, and shared memory (grad students).

Deliverables. All source code is to be written in C. Multiple source files must be used. At the very least, you will have:

- `db.c`, which will contain the `main` method. The final executable will be called `db`
- `menu.c`
- `even.c`

- `odd.c`

Other source files can be included if you like.

You will also need a `makefile` to compile your code. If you are using PTP, then you can use the Managed Makefile approach and allow the PTP plugin to produce the `makefile` for you.

You MUST also include a README text file to indicate which parts of the project work or don't work. The program is the most complex of your course submissions. If some things don't work properly or require explanation, please add this information here. Otherwise, you risk getting a lower grade than you deserve.

Final Submission: All files should then be combined into a single zip file as indicated below and submitted to Moodle.

LastName_FirstName_project.zip **(e.g., Smith_John_project.zip)**