Superlinear Speedup in Parallel Computation

Jing Shan

jshan@ccs.neu.edu

1 Introduction to The Problem

Because of its good speedup, parallel computing becomes more and more important in scientific computations, especially in those involving large-scaled data. When talking about the necessity, one claim is that we can always double the speed of chip every 18 months according to Moore’s Law, which means there is no need to develop parallel computation. This claim is proved to be wrong [1].

Speedup of a parallel computation is defined as $Sp = T/Tp$ [2], where $T$ is the sequential time of a problem and $Tp$ is the parallel time to solve the same problem using $p$ processors. $Tp$ was argued to be no greater than $P$ in [3]. However, in practice, people observed “superlinear speedup”, i.e. the speedup with $P$ processors is greater than $P$. Two main reasons for superlinear speedup are shown in [4]. The first reason is that in search problems, the termination time can be reduced when several searches are executed at the same time. And the second one is because of more efficient utilization of resources by multiprocessors. Our experiments focus on this aspect.

2 General Idea of Superlinear Speedup

Although parallel computation costs time in inter processor communications while sequential computation doesn’t, parallel computation can still achieve super linear speedup by utilizing resources more efficiently. One example is the reduction of RAM access time. This happens when the working set of a problem is greater than the cache size when executed sequentially, but can fit nicely in each available cache when executed in parallel. This leads to my first experiment in a shared memory multi-processor model and the second experiment in a distributed system.

Setting up data nodes is another experiment for speedup in distributed model. By partitioning the data in such a way that all data are fitted in caches of multiple data nodes, we can achieve a faster RAM access in a distributed system. However, this has a high requirement for Ethernet speed. I will describe this later.

3 Experiments Design and Implementation

3.1 Shared Memory Solution: 4 Processors Vs 1 Processor
My first experiment is to test the superlinear speedup in a shared memory system. In this experiment, I use a large array as the data and each node is trying to access a certain element of this array in 2 passes. Since the array element is the same size as on chip cache, the sequential execution will have a cache miss each time next element is read. While in parallel version, each element just fits in the L1 cache, so after the initialization, the data will be right in cache, hence there is no cache miss.

`cache.c` tests the cache size of current processor. The result I get is 16K for both joulian.hpcl and denali.ccs. `test1.c` is the main test code of array access written in TOPC. I use 4 nodes in this test and each of them need to access those elements with index = rank * 10 e.g. node1 need to access array[10], node2 needs to access array[20], node3 reads array[30] and node4 reads array[40]. This operation is repeated twice in this sequence.

The first row in table 1 is the elapsed time when all 4 slaves are running in node1e@joulian. The one in table 2 is the elapsed time when assign node1e to slave1, node2e to slave2, node3e to slave3, node4e to slave4 and pilgrim master. The second rows are read and write computation time and the third rows are data access time. The third row is where we should observe superlinear speedup.

In the first version of this code, I didn’t consider the read and write computation time, so the speedup is less than linear. Since each element of the array is an integer array called element[] of size 16K and what I did in the program is to assign a value to each cell of element array. So the read/write time is not ignorable.

Although the data access time has a very good speedup after considering the read and write time, I think there is something wrong with this result. The only difference between table 1 and table 2 is the number of slave nodes. In table 1, all 4 processes run in one node, while in table 2, they run in 4 difference nodes, so the read and write time should be the same for both tables. While in my result, they have 1.63 times difference. Since I use the system time instead of CPU time to calculate all three kinds of time here, I guess there may be some other system overheads in the sequential situation. I will change the time to CPU clock ticks soon and see what’s result.

<table>
<thead>
<tr>
<th></th>
<th>Exp1</th>
<th>Exp2</th>
<th>Exp3</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time</td>
<td>4.045049</td>
<td>3.797145</td>
<td>3.688709</td>
<td>3.84363</td>
</tr>
<tr>
<td>Read-write time</td>
<td>1.3941614</td>
<td>1.483788</td>
<td>1.327634</td>
<td>1.40186</td>
</tr>
<tr>
<td>Data access time</td>
<td>2.6508876</td>
<td>2.313357</td>
<td>2.361369</td>
<td>2.4418712</td>
</tr>
</tbody>
</table>

Table 1 **Sequential** (in second)
<table>
<thead>
<tr>
<th></th>
<th>Exp1</th>
<th>Exp2</th>
<th>Exp3</th>
<th>Exp4</th>
<th>Exp5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time</td>
<td>1.145449</td>
<td>1.130422</td>
<td>1.051195</td>
<td>1.108160</td>
<td>1.106291</td>
<td>1.1083034</td>
</tr>
<tr>
<td>Read-write time</td>
<td>0.856917</td>
<td>0.862434</td>
<td>0.862935</td>
<td>0.861918</td>
<td>0.862623</td>
<td>0.8613654</td>
</tr>
<tr>
<td>Data Access time</td>
<td>0.288532</td>
<td>0.267988</td>
<td>0.188260</td>
<td>0.246242</td>
<td>0.243668</td>
<td>0.246938</td>
</tr>
</tbody>
</table>

Table 2 4 nodes in parallel

<table>
<thead>
<tr>
<th></th>
<th>Avg. Sequential</th>
<th>Avg. 4 nodes</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time</td>
<td>3.84363</td>
<td>1.1083034</td>
<td>3.46</td>
</tr>
<tr>
<td>Read-write time</td>
<td>1.40186</td>
<td>0.8613654</td>
<td>1.63 ??</td>
</tr>
<tr>
<td>Access time</td>
<td>2.44187</td>
<td>0.246928</td>
<td>9.89</td>
</tr>
</tbody>
</table>

Table 3 Speedup

3.2 Distributed Systems: 4 Machine Vs One Machine

The code for this experiment is almost of the same as the shared memory model. The difference is instead of considering the time reading the large data, I read a small part of the data many times. I selected 4 Sun Blade100 machines in ccs. They are achird, acrux, acubens, adhafera. First, I run my TOPC code with 4 slaves all in one machine. And then I distributed them in 4 machines. The following two tables, table 4 and table 5 describe the result I got. And table 6 is the speedup in distributed situation.

Table 4 describes the result when I run my code with 4 slaves in one machine. The first row is the machines I tested separately and the second is the elapsed time in each machine. The last column is the average running time of these 4 tests.

Table 5 describes the result with 4 slaves running in 4 different machines. I made 5 tests with the same code. Elapsed time for each test is listed in the second row of table 5.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Achird</th>
<th>Acrux</th>
<th>Acubens</th>
<th>Adhafera</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>2.257833</td>
<td>2.174629</td>
<td>2.292703</td>
<td>2.890875</td>
<td>2.404</td>
</tr>
</tbody>
</table>

Table 4 4 slaves in one machine

<table>
<thead>
<tr>
<th>Test #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time</td>
<td>1.505024</td>
<td>1.479812</td>
<td>1.476588</td>
<td>1.379303</td>
<td>1.558780</td>
<td>1.4799814</td>
</tr>
</tbody>
</table>

Table 5 4 slaves in 4 different machines
<table>
<thead>
<tr>
<th>4 slaves in one machine</th>
<th>2.404</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 slaves in 4 machines</td>
<td>1.4799814</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.62</td>
</tr>
</tbody>
</table>

Table 6 **Speedup in Distributed Model**

As we can see from table 6, the speedup is far from superlinear. It is not even linear. The reason for this, I think, is the slow Ethernet speed. In ccs machines, the Ethernet speed is 100M bps, which is about 80 ns per byte. Using the assignment code of com3200, we can see the RAM access speed is about 50 ns. Although we can keep every data in cache with slaves running in distributed system, the Ethernet latency slows down the whole program. So we can get a conclusion that in order to gain superlinear speedup in distributed system by reducing RAM access time, we need a fast network.

### 3.3 Distributed Data Nodes

In order to simplify the code written by user and to manage data more efficiently, we need to set up distributed data nodes. Part of my work is the implement of the data nodes using RPC. To make things simple, I assume all of the data are read-only.

The architecture and the relationship for our master slave computation model and data nodes are described as the following diagrams.

First, the master start the remote data nodes by calling `StartRemoteDataNode()`, after the remote node is started successfully, master will get an handle `clnt_handle` to identify that data node. The handle is type of `CLIENT *`. After that master need to call `InitRemoteData()` and pass the whole data and the number of data nodes to this function. `InitRemoteData()` will then calculate which part to store for each data node and pass the data, the start key for that data node and the size it is responsible for to data nodes. To make the diagram clear, I only draw one data node, but actually there should be as many data nodes as the master assigned. In my code, I hardcode `Node_Num = 4` to make the work simple.

```
TOPC_master{
    StartRemoteDataNode();
    InitializeData();
}
```

Diagram 1

Diagram 2 shows how slaves get the data they need. When a slave want to access data, it calls `GetData()` with the data information such as hash key as the argument. `GetData()` uses the data information to calculate which data node to contact, say it is node 1, and call RPC

```
Data, StartKey, size
```
function return_data_1() with handle of node i as argument. RPC then get the data from that data node through return_data_1_svc() function. After RPC get the data, it passes it back to GetData() and GetData returns that data to slave.

```c
TOPC_slave{
    Result = GetData(data_info);
}
```

Diagram 2

The following are APIs for TOPC code and they are declared in `data.h` and implemented in `data.c` and `server.c`. Process “server”, which is generated from `server.c`, is running all the time in remote data node as a background program after it is started and it responds to RPC calls.

```c
void *StartRemoteNodes(char* host, char* prg, CLIENT ** cln);
/** This function is called by master and starts the remote data nodes **/
/** host is the machine name that acts as one data node *
* prg specify where is the server program located,
  e.g. ’/home/jshan/com3620/topc/server’ *
* cln is actually a return value for this function and it returns the client* handle to master **/
```

The code will call RPC function clnt_create() to create the data node in remote site.

```c
void *InitRemoteData(arrayElementType * data, CLIENT* clnts[Node_Num]);
/** This function is also called by master. It gets data and the array of client_handle from master, calculate which part belongs to which data node and then send the partition information to each data node. **/

/** data is a data buffer for the whole data. It should be type of DATA_TYPE, instead of arrayElementType. I hardcoded it this way to make the code easier.
* cln will be given a value of array of point to CLIENT. StartRemoteNodes only generates each client handle. It is in this function that all client handles are collected together. **/```
This function calls RPC code create_data_node_1() to send data to remote data nodes. The data it passed is type of DATA_INFO, which are defined in datanode.h. I will talk about this later. The actual function in server side is create_data_node_1_svc(), which will promote the remote node i.e. data server to initialize the remote side.

**RETURN_DATA_TYPE** *GetData(int key, int size);

/** *key* is the index for the data that you need to read.  
 * size is the data buffer size that need to be returned  
 * Currently RETURN_DATA_TYPE is defined as int *  
 ***/

This function calls RPC function return_data_1() to get the needed data.

**void** *DelRemoteNodes(CLIENT * clnt);

/* This function will call del_node_1() to stop the remote server process. */

The following are struct definitions and RPC functions declared in datanode.h and implemented in server.c datanode_xdr.c datanode_svc.c datanode_clnt.c.

**struct arrayElement{**
  int *data;
  int *start_key;
  **}**

/* This arrayElement struct is the same as the one in 3.1 and 3.2 */

**struct DATA_INFO {**
  arrayElementType *data;
  int *start_key;
  int *size;
  **}**

/* This is the parameter type used to pass data among RPC calls */

**typedef int** *RETURN_DATA_TYPE;

/* To make the code simple, I use int * as the return data type from data node */
int * create_data_node_1(DATA_INFO *, CLIENT *);
int * create_data_node_1_svc(DATA_INFO *, struct svc_req *);

/* These two functions implement the create_data() function in client and server side*/

RETURN_DATA_TYPE * return_data_1(int *, CLIENT *);
RETURN_DATA_TYPE * return_data_1_svc(int *, struct svc_req *);

/* These two implements return_data() function in client and server sides*/

void * del_node_1(void *, CLIENT *);
void * del_node_1_svc(void *, struct svc_req *);

/* These implements del_remote_data()*/

Currently, the RPC part is finished and it works well. Master can initialize data and pass data successfully. Given a client_handle, slave can pass and get data. But the whole work has not been finished yet because I don’t have enough time to implement all of these by myself in one quarter. The code to pass client_handle from master to slave hasn’t been finished. Since master and slaves are different processes and for each variable, they have their own copies, they can not shared the client_handle as a global data. So this handle needs to be passed explicitly between them.

4 Other Possible Interesting Test

In addition to the above discuss, there is another definition to speedup in [7]. Instead of comparing the execution time of fixed size problem, it is said we should use P times as large data in p nodes as the one in sequential situation and compare these time. If they are the same, it is a linear speedup and if parallel one has a shorter time, then it is superlinear speedup. I think this makes more sense in search problem, not in reducing RAM access time because increasing the problem size to us means either increasing the size in each data node or increasing the data node number. As the first, to increase the data size in each node, we can not get a better result because currently, our data has already filled cache. Increasing the size means cache miss in data node. As the second, to increase the data node number, I don’t think it will help a lot, either. Because when we increase the data node number, we will also increase the data size that needs to be send through the Ethernet. And make the data node number too big will cause more Ethernet latency. So I guess for search problem, it will give a better speedup.

5 Division of Labor

All of the work discussed above are done by myself.
References


