Shared Memory Programming

(Parallel Sorting by Regular Sampling)

Saeed Sarabchi *sarabchi@ualberta.ca*
Dept. name of *Computing Science, University of Alberta*

Abstract— In this technical report, the details relating to implementing Parallel Sorting from Regular Sampling algorithm would be elaborated. Plus, some experimentation in the performance and speedup will be provided. The goal is to assess performance improvements when Shared Programming style is used, specifically, SMDI style programming.

I. Introduction

In this project, PSRS Algorithm is implemented in C programming language on Ubuntu 14.04 operating system. The coding environment was eclipse. The machine configuration that the tests were run on was a 64bit with 2 CPU cores 2893 MHZ with 4 Gigabyte RAM.

Throughout this report, the implementation details are going to be discussed. In the experimentation section the execution time of Shared Memory PSRS algorithm will be investigated in detail and phase by phase. Moreover, the speedup graphs with variable input size will be elaborated.

II. Implementation Detail

The algorithm used in this project is based on [1]. The algorithm uses 4 phases to sort an array of keys. After creating the threads and dividing the input data evenly on the threads, the first Phase is to sort local data by quick sort. In the second phase, regular samples from all of the threads are gathered and sorted to find pivots in the same quantity as the threads. In the third phase, the data is partitioned in order for each thread to have a chunk of data with the highest locality. In the last phase the partial sorted lists of data in each thread should be merged (with O(n)) and then if we concatenate the thread data, the sorted input data would be the result. Before presenting each phase' code snippet, the main functions that are used in this implementation is listed and defined briefly:

- **Main** function for invoking the threads.
- **MySPMDMain** for the thread implementation including all four phases.
- **FillRegularSample** function to gather sampling data from all of the thread data which is used in phase 1 and filling out RegularSample Array.
- **ExtractPivot** function which is used in phase 2.
- **FillPartition** function to partition the local data I order to be assigned to their related thread where high locality data are then gathered and will concatenate with eachother in order to form the whole result.
- **MergePartialOrderedList** in order to merge the partitioned data that is transferred to each thread.

In the Code Snippet below, the structure of each thread (which is MySPMDMain function) is coded.

```
void * mySPMDMain( void * arg )
{
	// Parameters
	int counter;
	int myId;
	struct ThreadControlBlock* myTCB;
	myTCB = (struct ThreadControlBlock *)arg;
	myId = myTCB->id;

	pthread_barrier_wait (&barrier);

	//for capturing start and end time
	//startTiming
	MASTER
	{
		gettimeofday(&start, NULL);
	}

	// Phase 1 :
	// Sorting Chunks of Data which is Related to the Process

	int StartIndex=0;
	int LocalSize = NUM_GLOBALINPUTDATA/NUM_THREADS;
	StartIndex = myId * LocalSize;
	int EndIndex=0;

	if(myId == NUM_THREADS-1)
		EndIndex = NUM_GLOBALINPUTDATA-1;
	else
		EndIndex = StartIndex + LocalSize -1;

	qsort(&GlobalInputData[StartIndex],EndIndex-StartIndex+1, sizeof(int),
	compare);

	// Doing the Regular Sampling
	FillRegularSample(RegularSample, GlobalInputData, myId, StartIndex,
	EndIndex);

	// End of Phase 1

	pthread_barrier_wait (&barrier);
```

```
	/*
	Phase 2
	Sorting Regular Sample and Extracting Pivots
	*/
	MASTER
	{
	qsort(RegularSample, NUM_THREADS*NUM_THREADS, sizeof(int),
	compare);

	ExtractPivots(Pivots, RegularSample);
	}
	// End of Phase 2

	pthread_barrier_wait (&barrier);

	/*
	Phase 3 :
	Partitioning the data and
	because we are doing Shared Memory Programming,
	the Exchanging part is excluded
	*/
	FillPartition(Pivots, GlobalInputData, myId, StartIndex, EndIndex);

	pthread_barrier_wait (&barrier);

	/*
	Phase 4
	Merging the Partitions by Merging Partial OrderedLists to a single
	LinkedList
	*/
	MergePartialOrderedLists(GlobalInputData, myId);

	pthread_barrier_wait (&barrier);

	//EndTiming();
	MASTER
	{
		gettimeofday(&stop, NULL);
		unsigned long int  Duration= stop.tv_usec-start.tv_usec;
		printf("\n took : \n %ulli Mico sec , \n %lli Mili sec \n, %ulli
		Sec  to complete\n",Duration,
		Duration/1000,Duration/1000000);
	}
	return NULL;
}
```

In **MergePartialOrderedList** functions, the partial lists are merged in O(n) by traversing on all of the lists and maintaining a pinter to each list. Moreover, in order to save the sorted list, they are inserted into a linkedlist to be concatenated with each other:

```
	void MergePartialOrderedLists(int* InputDataArray, int ThreadID)
{
	int LocalSize=NUM_GLOBALINPUTDATA/NUM_THREADS;
	int counter;
	int StartIndex;
	int EndIndex;
	int DataCounter;
	int ElementCount=0;
	int StartEnds[NUM_THREADS][2];
	int buffer[NUM_THREADS];
	int PointersToThreadData[NUM_THREADS];
	for (counter=0; counter<NUM_THREADS; counter++)
	{
		if(ThreadID!=0)

			StartIndex=Partitions[counter][ThreadID-1];


		else
			StartIndex = counter * LocalSize;

		if(ThreadID!=NUM_THREADS-1)
		{
			EndIndex=Partitions[counter][ThreadID]-1;
		}
		else

			EndIndex=NUM_GLOBALINPUTDATA-1;

		StartEnds[counter][0]=StartIndex;
		StartEnds[counter][1]=EndIndex;
		buffer[counter]=InputDataArray[StartIndex];
		PointersToThreadData[counter]=StartIndex;
		ElementCount = ElementCount+(EndIndex-StartIndex)+1;
	}

	int MinIndex;
	SortedThreadDataLinkList[ThreadID]= malloc(sizeof(LinkListNode_t));
	LinkListNode_t*
	CurrentLinkListPointer=SortedThreadDataLinkList[ThreadID];

	for (counter=0; counter<ElementCount; counter++)
	{
	MinIndex=FindMinIndex(buffer);
		CurrentLinkListPointer->value=
	InputDataArray[PointersToThreadData[MinIndex]];
		CurrentLinkListPointer->Next=
	malloc(sizeof(LinkListNode_t));
	CurrentLinkListPointer = CurrentLinkListPointer->Next;
		PointersToThreadData[MinIndex]++;

	if(StartEnds[MinIndex][1]>=PointersToThreadData[MinIndex])

	buffer[MinIndex]=InputDataArray[PointersToThreadData[MinIndex]];
		else
			buffer[MinIndex]= MAX_NUMBER;
	}
}
```

III. Experimentation

In order to find out about the performance of the execution of Shared programming, this implementation has executed with number of inputs and threads. In the table below, the speedups are shown:

Input Size	#Threads	Execution Time	Speedup
8000000	1	717	1
8000000	2	490	1.463265306
8000000	3	220	3.259090909
800000	1	313	1
800000	2	270	1.159259259
800000	3	232	1.349137931

IV. References

[1] Xiabo Li, Paul Lu, Jonathan Schaeffer, John Shilington, Pok Sze Wong, Hanmao Shi, “On the versatility of parallel sorting by regular sampling”, Parallel Computing, Volume 19, Issue 10, October 1993, Pages 1079-1103.