Multicore Task

Parallelizing a few sorting algorithms

Submitted by: Akarsh Hegde

Sourabh S Shenoy

Submitted to: Prof. Raju

1 Abstract:

This report describes serial and parallel implementations of several sorting algorithms on a machine running Ubuntu 14.04 (on a VM, host being Windows 8.1). The aim of this project is to compare and discuss the performance of serial and parallel implementations of Merge, Bubble and Shell sort, in terms of time complexity.

2 System:

OpenMP on Ubuntu 14.04 running on Vmware Workstation, host: Windows 8.1

3 Algorithms and Code Snippets:

3.1 Merge Sort

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort.

3.1.1 Pseudocode

MERGE (*A*, *p*, *q*, *r*)

```
n1 ← q − p + 1
n2 ← r − q
Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
FOR i ← 1 TO n1
    DO L[i] ← A[p + i − 1]
FOR j ← 1 TO n2
    DO R[j] ← A[q + j ]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
FOR k ← p TO r
    DO IF L[i ] ≤ R[ j]
        THEN A[k] ← L[i]
            i ← i + 1
        ELSE A[k] ← R[j]
            j ← j + 1
```

3.1.2 Serial Merge Sort

```
void mergeSort(int arr[],int low,int mid,int high) {
    int i,m,k,l,temp[1000];
    l=low;
    i=low;
    m=mid+1;
    while((l<=mid)&&(m<=high)) {
        if(arr[l]<=arr[m]) {
            temp[i]=arr[l];
            l++;
        } else {
            temp[i]=arr[m];
            m++;
        }
        i++;
    }
    if(l>mid) {
        for(k=m;k<=high;k++) {
            temp[i]=arr[k];
            i++;
        }
    } else {
        for(k=l;k<=mid;k++) {
            temp[i]=arr[k];
            i++;
        }
    }
    for(k=low;k<=high;k++) {
        arr[k]=temp[k];
    }
}
```

```
void partition(int arr[],int low,int high) {
    int mid;
    if(low<high) {
        mid=(low+high)/2;
        partition(arr,low,mid);
        partition(arr,mid+1,high);
        mergeSort(arr,low,mid,high);
    }
}
```

3.1.3 Parallel Merge Sort

```
void pmerge(int i, int j) {
int mid = (i+j)/2;
int ai = i;
int bi = mid+1;
int newa[j-i+1], newai = 0;
while(ai <= mid && bi <= j) {
    if (array5[ai] > array5[bi])
        newa[newai++] = array5[bi++];
    else
        newa[newai++] = array5[ai++];
}
while(ai <= mid) {
    newa[newai++] = array5[ai++];
}
while(bi <= j) {
    newa[newai++] = array5[bi++];
}
for (ai = 0; ai < (j-i+1) ; ai++)
    array5[i+ai] = newa[ai];
}
```

```
void * pmergesort(void *a)
{
        NODE *p = (NODE *)a;
        NODE n1, n2;
        int mid = (p->i+p->j)/2;
        pthread_t tid1, tid2;
        int ret;
        n1.i = p->i;
        n1.j = mid;
        n2.i = mid+1;
        n2.j = p->j;
        if (p->i >= p->j) return;
        ret = pthread_create(&tid1, NULL, pmergesort, &n1);
        if (ret) {
            printf("%d %s - unable to create thread - ret - %d\n", __LINE__, __FUNCTION__, ret);
            exit(1);
        }
        ret = pthread_create(&tid2, NULL, pmergesort, &n2);
        if (ret) {
            printf("%d %s - unable to create thread - ret - %d\n", __LINE__, __FUNCTION__, ret);
            exit(1);
        }
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
        pmerge(p->i, p->j);
        pthread_exit(NULL);

}
```

3.2 Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

3.2.1 Pseudocode

```
procedure bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
     swapped = false
     for i = 1 to n-1 inclusive do
       if A[i-1] > A[i] then
         swap(A[i-1], A[i])
         swapped = true
       end if
     end for
     n = n - 1
   until not swapped
end procedure
```

3.2.2 Serial Bubble Sort

```
bubblesort() {
        int c,d,swap;
        for (c = 0 ; c < ( n - 1 ); c++)  {
           for (d = 0 ; d < n - c - 1; d++) {
                      if (array1[d] > array1[d+1])      {
                                swap = array1[d];
                                array1[d] = array1[d+1];
                                array1[d+1] = swap;
```

```
                }
            }
        }
        printf("\nBubble sort results\n");
        for ( c = 0 ; c < n ; c++ )
            printf("%d ", array1[c]);
        printf ("\n");
}
```

3.2.3 Parallel Bubble Sort

```
void bubblep() {
    int i, tmp, changes;
    int chunk;
    chunk = n / 2;
        changes = 1;
        int nr = 0;
            while(changes)   {
                #pragma omp parallel private(tmp)
                {
                    nr++;
                    changes = 0;
                    #pragma omp for reduction(+:changes)
                    for(i = 0; i < n - 1; i = i + 2) {
                        if(array4[i] > array4[i+1] ) {
                            tmp = array4[i];
                            array4[i] = array4[i+1];
                            array4[i+1] = tmp;
                            ++changes;
                        }
                    }
                    #pragma omp for reduction(+:changes)
                    for(i = 1; i < n - 1; i = i + 2) {
```

```
                if( array4[i] > array4[i+1] ) {
                    tmp = array4[i];
                    array4[i] = array4[i+1];
                    array4[i+1] = tmp;
                    ++changes;
                }
            }
        }
    }
    printf("\nBubble Sort with threads results\n");
    for ( i = 0 ; i < n ; i++ )
    printf("%d ", array4[i]);
    printf ("\n");
    return;
}
```

3.3 Shell Sort

Shell Sort, also known as Shell sort or Shell's method, is an in-place comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).

3.3.1 Pseudocode

```
span = int(n/2)
while span > 0 do:
for i = span .. n - 1 do:
  key =  num[i]
  j = i
  while j = span and num[j - span] > key do:
   num[j] = num[j - span]
   j = j - span
  num[j] = key
 span = int(span / 2.2)
```

3.3.2 Serial Shell Sort

```
void shellSort(int numbers[], int array_size) {

int i, j, increment, temp;
increment = 3;

while (increment > 0) {
	for (i=0; i < array_size; i++) {
		j = i;
		temp = numbers[i];

		while ((j >= increment) && (numbers[j-increment] > temp)) {
			numbers[j] = numbers[j - increment];
			j = j - increment;
		}

		numbers[j] = temp;
	}
	if (increment/2 != 0)
		increment = increment/2;
	else if (increment == 1)
		increment = 0;
	else
		increment = 1;
}
}
```

3.3.3 Parallel Shell Sort

```
void Shsort(int a[], int n, int s) {
    int j,i,key;
    for (j=s; j<n; j+=s) {
        key = a[j];
        i = j - s;
        while (i >= 0 && a[i] > key) {
            a[i+s] = a[i];
            i-=s;
        }
        a[i+s] = key;
    }
}

void shellsortp(int a[],int n) {
    int i, m;
    for(m = n/2; m > 0; m /= 2)
    {
        #pragma omp parallel for shared(a,m,n) private (i) default(none)
            for(i = 0; i < m; i++)
                Shsort(&(a[i]), n-i, m);
    }
    return;
}
```

4 Output

```
Before Sorting:

83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93
37 98 24 15 70

Bubble sort results
2 11 11 15 15 19 21 21 22 23 24 26 26 27 29 29 29 30 35 35 36 37 40 42 49 56 58 59 62 62 63 67 67 67 68 69 70
90 92 93 93 98

Time taken: 0.000036

Bubble Sort with threads results
2 11 11 15 15 19 21 21 22 23 24 26 26 27 29 29 29 30 35 35 36 37 40 42 49 56 58 59 62 62 63 67 67 67 68 69 70
90 92 93 93 98

Time taken: 0.000379

Merge Sort results
2 11 11 15 15 19 21 21 22 23 24 26 26 27 29 29 29 30 35 35 36 37 40 42 49 56 58 59 62 62 63 67 67 67 68 69 70
90 92 93 93 98
Time taken: 0.000030

Merge Sort with threads results
2 11 11 15 15 19 21 21 22 23 24 26 26 27 29 29 29 30 35 35 36 37 40 42 49 56 58 59 62 62 63 67 67 67 68 69 70
90 92 93 93 98
Time taken: 0.163386

Shell Sort results
2 11 11 15 15 19 21 21 22 23 24 26 26 27 29 29 29 30 35 35 36 37 40 42 49 56 58 59 62 62 63 67 67 67 68 69 70
90 92 93 93 98
Time taken: 0.000008

Shell Sort with threads results
2 11 11 15 15 19 21 21 22 23 24 26 26 27 29 29 29 30 35 35 36 37 40 42 49 56 58 59 62 62 63 67 67 67 68 69 70
90 92 93 93 98
Time taken: 0.000075

student@CSISLab01:~$
```

5 Conclusion

As observed from the implementation and the results thus obtained, we can infer that serial implementation in this case is more efficient than parallel implementation for these sorting algorithms. However, this should not be taken as a generalization as some other parallel implementation of these sorting techniques may prove to be faster than their serial counterparts.