Parallel and Distributed Systems Assignment 3

Sayak Chakraborti NetID-schakr11

March 2017

Compilation and Running

For cilk,
Compile-gcc guass_cilk.c -fcilkplus -lcilkrts -O3 -o cilk -w
Run-./cilk matrix_2000.dat
For openMP,
Compile-gcc guass_p.c -fopenmp -o openmp -O3 -w
Run-./openmp matrix_2000.dat

Introduction

In this assignment we are supposed to use the Guass-Jordan Elimination method for solving a set of linear equation using pivoting.The sequential code was already provided and we were supposed to parallelize a portion of the code in computeGuass function using openMP and cilk and compare the performance of the two.But first let us look at what this method is .
The system of linear equations is placed into a matrix with the right hand side representing the result and the left hand size has a matrix with coefficient and a the variable matrix. So Matrix*X=RHS, the objective is to find suitable values for the variables in matrix X so that they satisfy all the give equations.In order to achieve this we need to reduce the matrix along with the corresponding RHS to row-echelon form.There are three basic operations that we can perform to get to row-echelon form:-
1)Interchange Row(i) with Row(j),this needs to be done for RHS as well
2)Replace Row(i) with some scalar 'a' multiplied to Row(i) resulting in a.Row(i), this needs to be done for RHS as well
3)Replace Row(i) with Row(i)+a.Row(j),, this needs to be done for RHS as well
Using these three rules in repetitive format,we will achieve the reduced row-echelon form which will look like:-
1)The rows consisting of entirely 0's are grouped together at the bottom of the matrix.
2)In the rows that don't consists of entirely of 0's,the leftmost non-zero number is 1.(also known as the pivot element).

3)Each column that has this pivot element(1) has rest of the elements as 0s.
4)The pivot element(1) in any row is left of the pivot element(1) in any rows below.
In the code provided we using pivoting to make the element above and below a leading 1 to be zeros.We find the pivot value using the row with largest absolute value in the main diagonal.Getting which is sequential and in critical path and can't be parallelized.Once we get our pivot value,we check whether the pivot value is already 1 or not(remember we are trying to have leading 1's on the main diagonal),if not then we divide this entire row with the pivot value to make it one. **we can make the elements above and below this pivot value in this column into 0's by subtracting there column value times the pivot row from them,and this can be done in parallel and this is what I have parallelized.**This process continues over multiple iterations until we get our result that is the diagonal is 1 and we have result of our variable on the right hand side.This is a very small portion of the code that I have parallelized and run my experiments on.

OpenMP

OpenMP is an API that is mainly used for parallel programming by multithreading and shared memory operations.OpenMP doesn't care about whether the underlying shared memory structure is Uniform Memory Access or Non-Uniform Memory access.Parallelism is accomplished with the use of threads which share a single process resource and thus the address space.To reduce cache line bouncing between different cores ,some variables that are not shared can be localized to the threads by declaring them private when using the parallel portion.The pragma omp parallel is used with schedule clause with different scheduling techniques like static,dynamic,auto etc. As the assignment required us test different scheduling techniques like CYCLIC,DYNAMIC and BLOCKED,I used static with chunk size 1 for CYCLIC scheduling and Static with variable chunk size for BLOCK based scheduling .Since I was only concerned to parallelise a loop hence the schedule clause worked for me.I also used auto,runtime and guided option to get some results. I will briefly try to explain what I understood these options as:-
1)Static-divides the loop into equal chunks as close as possible.
2)dynamic-start with some chunk sized block ,when finished take blcks from work queue which keeps track of the loop iterations.
3)guided-chunk size start of large but decrease gradually to provide better performance.
4)auto-decision regarding loop scheduling is delegated to compiler.
5)runtime-takes the value set by environment variable(by default maybe dynamic,not sure though).

Cilk

Cilk is a language extension to C and C++ provided by Intel that is used for parallel programs.For parallelising loops cilk_for is much better than using cilk_spawn with cilk_sync as it has less overheads.But we have to keep in mind that these are opportunities for parallelism and not mandates,it is the runtime environment that decides whether and to what extent parallelism will occur.I have used __cilkrts_set_param to set the number of threads.I also changed the grainsize to see what effect it had.

Observations

Some generic observations were,executing the code with different thread numbers for the same matrix one after another causes better performance due to locality.Better partitioning causes better performance sometimes get a thread value that equally divides work.Serialization part remains near constant time requirement.Reduction operation combining the results computed by different threads has been considered serial for this experiment.
Declare independent variables as private so that each thread has separate copies of them rather than shared copied,because accessing shared memory requires overheads like cache coherence which causes excess latency.
For openMP in most the execution time decreases with increase in number of threads linearly upto a certain number of threads which is in the ball park of 40 and I observe severe spike in execution time from 40 to 72.This may be due to the workload not being large enough to get each thread appropriate job and more threads means more context switches,preemption and contenting for the shared variables which would induce high traffic and cause high latency.The critical path in the code of finding the pivotval required access of shared variables and this may cause contention.Also for higher number of threads there are additional costs of book keeping and synchronization overhead as number of threads increase. Also multiple cache lines which are in the critical path bounce back and forth due to cache coherence.
As the number of threads become more than the number of hardware cores the execution time comes back to normal levels and this may be due to the smart scheduling by the scheduler which schedules the threads in such a way that exploits cache locality.More hardware threads means more cache ping pong, more threads than hardware then scheduler has job to do.
Cilk has better scheduling and less overhead for which reason it outperforms openMP in every way.Also among the different scheduling mechanisms of openMP due to the scare level of parallelism induced in the program the performance of each of them is nearly comparable.Auto schedule has the best speed up for the openMP version but still the cilk versions have better consistency and don't suffer at the range of 40-72 threads where the openMP version seem to flat out.

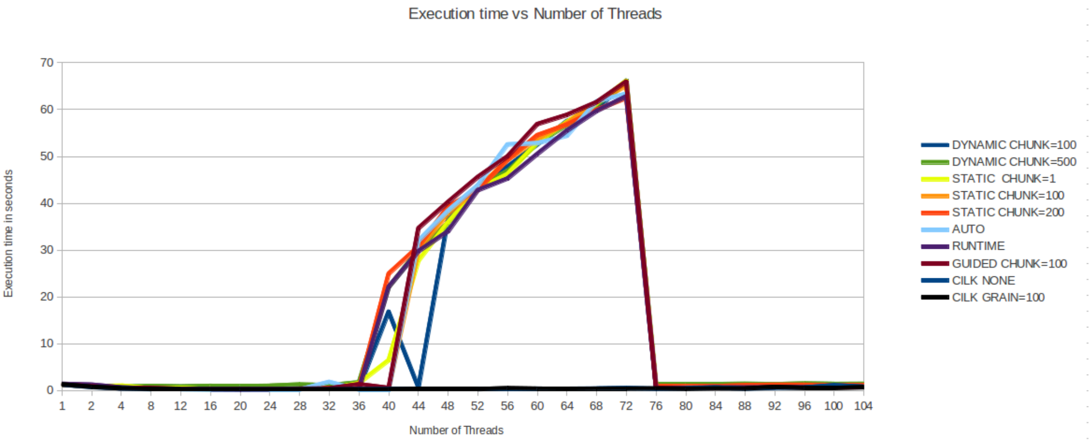
Experimentation

The experiments here have been run on node2x18a and for the matrix_2000 benchmark.I have used O3 optimization in each build for these experiments. I observe that cilk out-performs any openMP implementation.For openMP implementations there is a high spike of execution time when the number of threads are increased to approximately 40 and the spike stays till the number of threads become more than the number of core(72) after which it again goes back to lower execution time.This is not the case for cilk which doesn't experience such spikes.

Also for CYCLIC job allocation I have used the static flag with chunk size 1 and for BLOCKED allocation I have used static with variable block size.For DYNAMIC I have used the dynamic flag with variable chunk size.I have observed that increasing the block size increases the execution time(that is it reduces performance),because for higher number of threads higher block size means for this particular benchmark not enough work that can be distributed to the threads equally.Also for lower chunk size each thread has less work to do and thus can have higher parallelism.

The results included(execution time) have been determined with multiple iterative running for the same number of threads and the most frequently occurring value is taken with very negligible deviation.

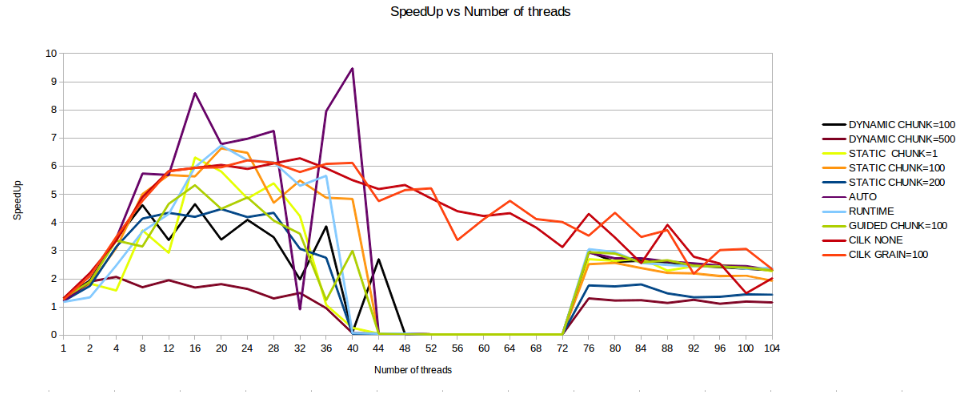
Number of threads	Matrix	Error					Time in seconds					
			DYNAMIC(100)	DYNAMIC(500)	CYCLIC	BLOCKED(100)	BLOCKED(200)	AUTO	RUNTIME	GUIDED(100)	CILK NONE	CILK(100)
1	2000	0.00E+000	1.288852	1.286953	1.291091	1.28789	1.399932	1.401145	1.434808	1.290354	1.289737	1.340651
2	2000	0.00E+000	0.874729	0.892438	0.927396	0.93355	0.971653	0.896795	1.264601	0.895684	0.768772	0.810627
4	2000	0.00E+000	0.498364	0.820984	1.07095	0.549861	0.541842	0.495068	0.686501	0.505608	0.507147	0.487477
8	2000	0.00E+000	0.367775	0.998092	0.456658	0.339098	0.410572	0.296286	0.462032	0.539622	0.346674	0.35639
12	2000	0.00E+000	0.503177	0.870464	0.581541	0.298968	0.391288	0.298917	0.395089	0.364458	0.291692	0.292183
16	2000	0.00E+000	0.365106	1.003926	0.26942	0.301511	0.404447	0.197744	0.284981	0.319195	0.286178	0.286061
20	2000	0.00E+000	0.500884	0.937117	0.292428	0.256318	0.379462	0.250569	0.25237	0.378879	0.281218	0.285152
24	2000	0.00E+000	0.414993	1.034089	0.350112	0.262529	0.405394	0.243748	0.273704	0.347563	0.287961	0.273898
28	2000	0.00E+000	0.488575	1.302425	0.315243	0.361686	0.391221	0.23439	0.276866	0.417857	0.278736	0.27772
32	2000	0.00E+000	0.858234	1.134116	0.402046	0.309586	0.554019	1.862312	0.320487	0.471699	0.270679	0.293464
36	2000	0.00E+000	0.439653	1.768802	1.581309	0.348296	0.618357	0.213721	0.300362	1.361618	0.286938	0.279487
40	2000	0.00E+000	16.861782	22.085904	6.477251	0.351405	24.963685	0.179312	22.150969	0.56697	0.308805	0.277896
44	2000	0.00E+000	0.629706	30.018919	27.733183	29.994537	30.750672	31.972418	29.749986	34.657125	0.327615	0.357132
48	2000	0.00E+000	36.723338	35.836079	35.961736	37.742383	39.297675	38.419559	34.015608	40.26852	0.318815	0.329928
52	2000	0.00E+000	43.784333	43.802789	43.025569	43.142939	42.852183	43.823715	42.779396	45.596603	0.350228	0.325972
56	2000	0.00E+000	47.789305	46.819951	46.221177	49.244522	49.209327	52.557054	45.259597	49.98298	0.38612	0.503515
60	2000	0.00E+000	52.54255	52.759186	52.917308	53.62885	54.526439	52.827258	50.501371	56.889605	0.401786	0.412766
64	2000	0.00E+000	57.539519	56.72779	57.368672	57.109755	56.786587	54.407172	55.506105	58.873299	0.392395	0.356313
68	2000	0.00E+000	59.927495	60.987898	61.293436	61.559532	59.951037	61.760013	59.687199	61.585566	0.444778	0.412375
72	2000	0.00E+000	66.197033	65.66576	66.160675	65.102332	62.476442	63.378758	62.826367	65.979642	0.543621	0.423188
76	2000	0.00E+000	0.573068	1.300211	0.630032	0.67395	0.961675	0.579716	0.556425	0.576546	0.394613	0.481228
80	2000	0.00E+000	0.652556	1.379427	0.644059	0.662072	0.980745	0.620668	0.575908	0.586718	0.490049	0.391104
84	2000	0.00E+000	0.642371	1.367144	0.624659	0.712971	0.942292	0.621512	0.662829	0.659025	0.664326	0.487474
88	2000	0.00E+000	0.655755	1.48489	0.741689	0.768423	1.14456	0.648876	0.681642	0.638847	0.433467	0.455636
92	2000	0.00E+000	0.687744	1.352467	0.694584	0.774163	1.26304	0.666577	0.695094	0.688932	0.610201	0.779645
96	2000	0.00E+000	0.703195	1.521482	0.683142	0.810632	1.244961	0.688196	0.691538	0.698553	0.668352	0.562482
100	2000	0.00E+000	0.719391	1.421313	0.700147	0.80242	1.170762	0.693296	0.720565	0.708366	1.139416	0.554942
104	2000	0.00E+000	0.737648	1.460915	0.747546	0.879925	1.184699	0.736529	0.710642	0.738503	0.83925	0.727561



For speedup calculation I have ran the serial code and measured the execution time which comes at 1.70 seconds and used the formulae SpeedUp(p)=Time(1)/Time(p) ,where Time(1) refers to the execution of the code without any parallelism and Time(p) refers to the execution time with the parallel portion being run by p threads/processes.

The speedup results and curve are below:-

Number of threads	Matrix	SpeedUp									
		DYNAMIC(100)	DYNAMIC(500)	CYCLIC	BLOCKED(100)	BLOCKED(200)	AUTO	RUNTIME	GUIDED(100)	CILK NONE	CILK(100)
1	2000	1.319003	1.32095	1.316716	1.319989	1.214345	1.213293	1.184828	1.317468	1.318098	1.268041
2	2000	1.943459	1.904894	1.83309	1.821006	1.749596	1.895639	1.344298	1.897991	2.211319	2.097142
4	2000	3.411161	2.070686	1.587376	3.09169	3.137446	3.433872	2.476326	3.362289	3.352085	3.487344
8	2000	4.622391	1.70325	3.722698	5.0133	4.140565	5.737699	3.679399	3.150353	4.903742	4.770055
12	2000	3.378533	1.952981	2.923268	5.686227	4.344626	5.687197	4.302828	4.664461	5.828065	5.818271
16	2000	4.656182	1.693352	6.309851	5.638269	4.20327	8.596974	5.96531	5.325898	5.940359	5.942788
20	2000	3.393999	1.814074	5.813397	6.632386	4.480027	6.784558	6.736141	4.486921	6.045132	5.961733
24	2000	4.096455	1.643959	4.855589	6.475475	4.193451	6.974416	6.211089	4.8912	5.903577	6.20669
28	2000	3.479507	1.305258	5.392665	4.70021	4.34537	7.252869	6.140154	4.068377	6.098961	6.121273
32	2000	1.980812	1.498965	4.228372	5.491204	3.068487	0.912844	5.304427	3.603993	6.280502	5.792874
36	2000	3.866686	0.961102	1.075059	4.880906	2.749221	7.954296	5.659837	1.248515	5.924625	6.082573
40	2000	0.10082	0.076972	0.262457	4.837723	0.068099	9.480682	0.076746	2.998395	5.505092	6.117396
44	2000	2.699673	0.056631	0.061298	0.056677	0.055283	0.053171	0.057143	0.049052	5.189018	4.760145
48	2000	0.046292	0.047438	0.047272	0.045042	0.04326	0.044248	0.049977	0.042217	5.332246	5.152639
52	2000	0.038827	0.03881	0.039511	0.039404	0.039671	0.038792	0.039739	0.037283	4.853981	5.215172
56	2000	0.035573	0.036309	0.03678	0.034522	0.034546	0.032346	0.037561	0.034012	4.402776	3.376265
60	2000	0.032355	0.032222	0.032126	0.031699	0.031178	0.03218	0.033662	0.029882	4.231108	4.118556
64	2000	0.029545	0.029968	0.029633	0.029767	0.029937	0.031246	0.030627	0.028876	4.332369	4.771086
68	2000	0.028368	0.027874	0.027735	0.027616	0.028356	0.027526	0.028482	0.027604	3.822131	4.122461
72	2000	0.025681	0.025889	0.025695	0.026113	0.02721	0.026823	0.027059	0.025766	3.127179	4.017127
76	2000	2.966489	1.30748	2.698276	2.522442	1.767749	2.93247	3.055219	2.948594	4.308018	3.532629
80	2000	2.60514	1.232396	2.63951	2.567697	1.733376	2.738984	2.95186	2.897474	3.469041	4.34667
84	2000	2.646446	1.243468	2.721485	2.384389	1.804112	2.735265	2.564764	2.579568	2.558985	3.487365
88	2000	2.592432	1.144866	2.292066	2.212323	1.485287	2.619915	2.493978	2.661044	3.921867	3.731048
92	2000	2.47185	1.256962	2.447508	2.19592	1.345959	2.550343	2.445712	2.467588	2.785967	2.18048
96	2000	2.417537	1.117332	2.488502	2.097129	1.365505	2.470227	2.458289	2.433602	2.54357	3.022319
100	2000	2.36311	1.196077	2.428062	2.118591	1.452046	2.452055	2.35926	2.399889	1.491992	3.063383
104	2000	2.304622	1.163654	2.274108	1.931983	1.434964	2.308124	2.392203	2.301954	2.025618	2.336574


SpeedUp vs Number of threads
10
9
8
7
6
5
4
3
2
1
0
SpeedUp
1
2
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
68
72
76
80
84
88
92
96
100
104
Number of threads
DYNAMIC CHUNK=100
DYNAMIC CHUNK=500
STATIC CHUNK=1
STATIC CHUNK=100
STATIC CHUNK=200
AUTO
RUNTIME
GUIDED CHUNK=100
CILK NONE
CILK GRAIN=100