

Thapar Institute of Engineering and Technology

Parallel and Distributed Computing

LAB Assignment 1

Submitted by :

Bhavish Pushkarna

102303279

3C22

Submitted to

Dr. Saif Nalband

Introduction & Objective

The objective of this session was to quantify the efficiency of parallel execution strategies using the **OpenMP API** on multi core hardware. The study was designed to expose the limitations of parallelism by testing workloads against the **Roofline Model** specifically targeting compute bound versus memory bound regions.

We analyzed three specific kernel types:

- **Matrix Multiplication:** A high arithmetic intensity workload used to evaluate the trade offs between computational throughput and cache hit rates.
- **Monte Carlo π Estimation:** A low latency, high dependency free workload used to measure the theoretical peak scalability of the processor cores.
- **DAXPY:** A low arithmetic intensity vector loop used to purposefully saturate the memory bus and identify bandwidth ceilings.

The ultimate goal was to move beyond "speedup" as a raw number and analyze the architectural phenomena driving the results. This included evaluating **cache locality** optimizations, the impact of **thread creation overhead**, and the distinct performance behaviors observed when a system transitions from being compute limited to memory bandwidth limited.

Experiment 1: DAXPY Loop

Analyzing Scalability Limits and Thread Overhead.

Methodology

The DAXPY loop $y = \alpha x + y$ is a simple vector operation. The dataset size used was 2^{16} (65,536 elements).

- **Implementation:** We dynamically varied the number of threads from 2 to 12 to observe the scaling curve.

Code :

The screenshot shows a terminal window with the following details:

- Title bar: LAB1 x daxpy_loop.cpp x
- Code content:

```
1 #include <chrono>
1 #include <cstdlib>
2 #include <limits>
3 #include <omp.h>
4 #include <random>
5 #include <vector>
6 #include<iostream>
7 using namespace std;
8 double get_random_number(){
9     static thread_local mt19937 mt{random_device{}()};
10    static thread_local uniform_real_distribution<double> random_range{0.0,1.0};
11    return random_range(mt);
12 }
13
14 int main(int argc, char** argv){
15     if(argc < 2){
16         cout<<"Usage: ./daxpy_loop scalar_value"<<"\n";
17         return EXIT_FAILURE;
18     }
19     int SIZE{1<<16};
20     int total_num_threads{omp_get_max_threads()};
21     double a{atof(argv[1])};
22     vector<double> x(SIZE, 0.0);
23     vector<double> y(SIZE, 0.0);
24
25     for(int start_threads{2};start_threads <= total_num_threads; ++start_threads){
26         cout<<"Using threads : "<< start_threads<<"\n";
27         omp_set_num_threads(start_threads);
28         auto start_time{chrono::steady_clock::now()};
29         #pragma omp parallel for
30         for(int i = 0;i<SIZE;++i){
31             x[i] = get_random_number();
32             y[i] = get_random_number();
33             x[i] = a*x[i] + y[i];
34         }
35         auto end_time{chrono::steady_clock::now()};
36         auto exec_time{end_time - start_time};
37         chrono::duration<double, std::milli> ms{exec_time};
38         cout<<"Execution time: "<<ms.count()<<"\n";
39     }
40     return EXIT_SUCCESS;
41 }
```
- Status bar: NORMAL ➤ ↻ main > ~1 ➤ daxpy_loop.cpp utf-8 ⌂ ⌂ Top ⌂ 1:1

Terminal Output :

```
▲ all_might ➜ ~/Documents/UCS645/LAB1 git-[* main]- ➤ perf stat ./daxpy_loop 10.2
Using threads : 2
Execution time: 5.42209
Using threads : 3
Execution time: 3.85289
Using threads : 4
Execution time: 3.58243
Using threads : 5
Execution time: 3.58473
Using threads : 6
Execution time: 3.59785
Using threads : 7
Execution time: 3.50829
Using threads : 8
Execution time: 3.39349
Using threads : 9
Execution time: 3.34596
Using threads : 10
Execution time: 3.30003
Using threads : 11
Execution time: 2.98046
Using threads : 12
Execution time: 6.92014

Performance counter stats for './daxpy_loop 10.2':
      304,729,262      task-clock:u          #      3.956 CPUs utilized
                  0      context-switches:u      #      0.000 /sec
                  0      cpu-migrations:u      #      0.000 /sec
                 456      page-faults:u        #      1.496 K/sec
      246,240,944      instructions:u       #      0.24  insn per cycle
      1,043,629,884      cycles:u           #      3.425 GHz
      28,780,515      branches:u          #   94.446 M/sec
                 47,199      branch-misses:u     #      0.16% of all branches

      0.077038153 seconds time elapsed
      0.303086000 seconds user
      0.002981000 seconds sys

▲ all_might ➜ ~/Documents/UCS645/LAB1 git-[* main]- ➤ |
```

Performance Analysis :

Number of Threads	Execution Time (ms)	Trend/ Observation	Architectural Inference
2	5.42209	Baseline	Initial parallel overhead is amortized, but workload per thread is high.
3	3.85289	Significant Gain	A major jump in speed i.e 29% faster than 2 threads as work is divided further.
4 -6	3.58 3.59	Plateau	Performance stabilizes. The overhead of managing more threads starts to cancel out the benefits of parallelization.
7-10	3.50 3.30	Gradual Improvement	Slight improvements continue as the scheduler manages to squeeze more efficiency out of the CPU resources.
11	2.98046	Peak Performance	Optimal Operating Point. The scheduler found the perfect balance between computing power and thread management overhead.
12	6.92014	Performance Degradation	Oversubscription or Contention. The time doubles worse than 2 threads. This suggests resource contention cache thrashing or context switching overhead overwhelmed the small dataset.

Architectural Analysis :

This experiment reveals the **Limitations of Parallelism**:

1. **Thread Creation Overhead:** Spawning a thread takes CPU cycles. When the workload (65k elements) is small, the time taken to create and manage 12 threads exceeds the time saved by parallelizing the math.
2. **Memory Bandwidth Saturation:** This task is **Memory Bound**.

Proof: The perf stat shows an IPC of **0.24**. This is very low. It means the CPU is spending approximately 75% of its cycles *waiting* for data from RAM. Adding more threads doesn't help because the bottleneck is the wire to the RAM, not the CPU speed.

3. **Context Switching:** The spike at 12 threads 6.92ms suggests the system ran out of physical/logical cores, forcing the OS to "Context Switch" swap tasks rapidly, which ruins cache performance.

Experiment 2: Matrix Multiplication

Analyzing the effects of Cache Optimization and Loop Collapsing.

Methodology & Code Optimization

Matrix multiplication is traditionally an $O(N^3)$ operation. In the serial implementation, multiplying Row A by Column B causes a specific memory issue: **Strided Access**.

- **The Problem:** Matrices are stored in Row Major order. Accessing elements column wise in Matrix B leads to frequent **Cache Misses** because the CPU fetches a cache line but only uses one value from it.
- **The Solution (Implemented in matrix_multiplication.cpp):**
We implemented **Matrix Transposition** before the multiplication loop.
- C++
- **Why this matters:** This converts the operation into a Dot Product of two rows. The CPU can now read contiguous memory addresses, maximizing **Spatial Locality** and allowing the Prefetcher to keep the L1/L2 cache filled.

Code :

```
all_might      matrix_multiplication.cpp
36 #include <chrono>
35 #include <limits>
34 #include <random>
33 #include <vector>
32 #include<iostream>
31 using namespace std;
30 double get_random_number(){
29     static thread_local mt19937 mt{random_device{}()};
28     static thread_local uniform_real_distribution<double> random_range{0.0,1.0};
27     return random_range(mt);
26 }
25 int main(){
24     constexpr int ROW_SIZE{1000};
23     constexpr int COL_SIZE{1000};
22     vector<double> mat_a(ROW_SIZE*COL_SIZE, 0.0);
21     vector<double> mat_b_T(ROW_SIZE * COL_SIZE, 0.0);
20     vector<double> mat_b(ROW_SIZE*COL_SIZE, 0.0);
19     vector<double> res(ROW_SIZE*COL_SIZE, 0.0);
18
17
16     #pragma omp parallel for collapse(2)
15     for(int i = 0; i<ROW_SIZE; ++i){
14         for(int j = 0; j<COL_SIZE; ++j){
13             mat_a[i*COL_SIZE+j] = get_random_number();
12             mat_b[i*COL_SIZE+j] = get_random_number();
11         }
10
9
8
7     #pragma omp parallel for collapse(2)
6     for(int i = 0; i < ROW_SIZE; ++i){
5         for(int j = 0; j < COL_SIZE; ++j){
4             mat_b_T[j * ROW_SIZE + i] = mat_b[i * COL_SIZE + j];
3         }
2
1
37
1     cout<<"Using No Threading"<<"\n";
2     auto start_time{chrono::steady_clock::now()};
3     for(int i = 0; i<ROW_SIZE; ++i){
4         for(int j = 0; j<COL_SIZE; ++j){
5             double sum{0.0};
6             for(int k = 0; k<ROW_SIZE; ++k){
7                 sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
8             }
9             res[i*COL_SIZE+j] = sum;
10        }
11    }
12
13    cout<<"Time taken : "<<(chrono::steady_clock::now() - start_time).count()<<" ms\n";
14
15    cout<<"Matrix Multiplication Result :\n";
16    for(int i = 0; i<ROW_SIZE; ++i){
17        for(int j = 0; j<COL_SIZE; ++j){
18            cout<<res[i*COL_SIZE+j]<<" ";
19        }
20        cout<<"\n";
21    }
22
23    cout<<"\n";
24
25    cout<<"Matrix A :\n";
26    for(int i = 0; i<ROW_SIZE; ++i){
27        for(int j = 0; j<COL_SIZE; ++j){
28            cout<<mat_a[i*COL_SIZE+j]<<" ";
29        }
30        cout<<"\n";
31    }
32
33    cout<<"Matrix B :\n";
34    for(int i = 0; i<ROW_SIZE; ++i){
35        for(int j = 0; j<COL_SIZE; ++j){
36            cout<<mat_b_T[i*COL_SIZE+j]<<" ";
37        }
38        cout<<"\n";
39    }
40
41    cout<<"Matrix Transposed B :\n";
42    for(int i = 0; i<ROW_SIZE; ++i){
43        for(int j = 0; j<COL_SIZE; ++j){
44            cout<<mat_b[i*COL_SIZE+j]<<" ";
45        }
46        cout<<"\n";
47    }
48
49    cout<<"\n";
50
51    cout<<"Matrix Result :\n";
52    for(int i = 0; i<ROW_SIZE; ++i){
53        for(int j = 0; j<COL_SIZE; ++j){
54            cout<<res[i*COL_SIZE+j]<<" ";
55        }
56        cout<<"\n";
57    }
58
59    cout<<"\n";
60
61    cout<<"Matrix A :\n";
62    for(int i = 0; i<ROW_SIZE; ++i){
63        for(int j = 0; j<COL_SIZE; ++j){
64            cout<<mat_a[i*COL_SIZE+j]<<" ";
65        }
66        cout<<"\n";
67    }
68
69    cout<<"Matrix B :\n";
70    for(int i = 0; i<ROW_SIZE; ++i){
71        for(int j = 0; j<COL_SIZE; ++j){
72            cout<<mat_b_T[i*COL_SIZE+j]<<" ";
73        }
74        cout<<"\n";
75    }
76
77    cout<<"Matrix Transposed B :\n";
78    for(int i = 0; i<ROW_SIZE; ++i){
79        for(int j = 0; j<COL_SIZE; ++j){
80            cout<<mat_b[i*COL_SIZE+j]<<" ";
81        }
82        cout<<"\n";
83    }
84
85    cout<<"Matrix Result :\n";
86    for(int i = 0; i<ROW_SIZE; ++i){
87        for(int j = 0; j<COL_SIZE; ++j){
88            cout<<res[i*COL_SIZE+j]<<" ";
89        }
90        cout<<"\n";
91    }
92
93    cout<<"\n";
94
95    cout<<"Matrix A :\n";
96    for(int i = 0; i<ROW_SIZE; ++i){
97        for(int j = 0; j<COL_SIZE; ++j){
98            cout<<mat_a[i*COL_SIZE+j]<<" ";
99        }
100       cout<<"\n";
101      }
102
103      cout<<"Matrix B :\n";
104      for(int i = 0; i<ROW_SIZE; ++i){
105          for(int j = 0; j<COL_SIZE; ++j){
106              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
107          }
108          cout<<"\n";
109      }
110
111      cout<<"Matrix Transposed B :\n";
112      for(int i = 0; i<ROW_SIZE; ++i){
113          for(int j = 0; j<COL_SIZE; ++j){
114              cout<<mat_b[i*COL_SIZE+j]<<" ";
115          }
116          cout<<"\n";
117      }
118
119      cout<<"Matrix Result :\n";
120      for(int i = 0; i<ROW_SIZE; ++i){
121          for(int j = 0; j<COL_SIZE; ++j){
122              cout<<res[i*COL_SIZE+j]<<" ";
123          }
124          cout<<"\n";
125      }
126
127      cout<<"\n";
128
129      cout<<"Matrix A :\n";
130      for(int i = 0; i<ROW_SIZE; ++i){
131          for(int j = 0; j<COL_SIZE; ++j){
132              cout<<mat_a[i*COL_SIZE+j]<<" ";
133          }
134          cout<<"\n";
135      }
136
137      cout<<"Matrix B :\n";
138      for(int i = 0; i<ROW_SIZE; ++i){
139          for(int j = 0; j<COL_SIZE; ++j){
140              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
141          }
142          cout<<"\n";
143      }
144
145      cout<<"Matrix Transposed B :\n";
146      for(int i = 0; i<ROW_SIZE; ++i){
147          for(int j = 0; j<COL_SIZE; ++j){
148              cout<<mat_b[i*COL_SIZE+j]<<" ";
149          }
150          cout<<"\n";
151      }
152
153      cout<<"Matrix Result :\n";
154      for(int i = 0; i<ROW_SIZE; ++i){
155          for(int j = 0; j<COL_SIZE; ++j){
156              cout<<res[i*COL_SIZE+j]<<" ";
157          }
158          cout<<"\n";
159      }
160
161      cout<<"\n";
162
163      cout<<"Matrix A :\n";
164      for(int i = 0; i<ROW_SIZE; ++i){
165          for(int j = 0; j<COL_SIZE; ++j){
166              cout<<mat_a[i*COL_SIZE+j]<<" ";
167          }
168          cout<<"\n";
169      }
170
171      cout<<"Matrix B :\n";
172      for(int i = 0; i<ROW_SIZE; ++i){
173          for(int j = 0; j<COL_SIZE; ++j){
174              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
175          }
176          cout<<"\n";
177      }
178
179      cout<<"Matrix Transposed B :\n";
180      for(int i = 0; i<ROW_SIZE; ++i){
181          for(int j = 0; j<COL_SIZE; ++j){
182              cout<<mat_b[i*COL_SIZE+j]<<" ";
183          }
184          cout<<"\n";
185      }
186
187      cout<<"Matrix Result :\n";
188      for(int i = 0; i<ROW_SIZE; ++i){
189          for(int j = 0; j<COL_SIZE; ++j){
190              cout<<res[i*COL_SIZE+j]<<" ";
191          }
192          cout<<"\n";
193      }
194
195      cout<<"\n";
196
197      cout<<"Matrix A :\n";
198      for(int i = 0; i<ROW_SIZE; ++i){
199          for(int j = 0; j<COL_SIZE; ++j){
200              cout<<mat_a[i*COL_SIZE+j]<<" ";
201          }
202          cout<<"\n";
203      }
204
205      cout<<"Matrix B :\n";
206      for(int i = 0; i<ROW_SIZE; ++i){
207          for(int j = 0; j<COL_SIZE; ++j){
208              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
209          }
210          cout<<"\n";
211      }
212
213      cout<<"Matrix Transposed B :\n";
214      for(int i = 0; i<ROW_SIZE; ++i){
215          for(int j = 0; j<COL_SIZE; ++j){
216              cout<<mat_b[i*COL_SIZE+j]<<" ";
217          }
218          cout<<"\n";
219      }
220
221      cout<<"Matrix Result :\n";
222      for(int i = 0; i<ROW_SIZE; ++i){
223          for(int j = 0; j<COL_SIZE; ++j){
224              cout<<res[i*COL_SIZE+j]<<" ";
225          }
226          cout<<"\n";
227      }
228
229      cout<<"\n";
230
231      cout<<"Matrix A :\n";
232      for(int i = 0; i<ROW_SIZE; ++i){
233          for(int j = 0; j<COL_SIZE; ++j){
234              cout<<mat_a[i*COL_SIZE+j]<<" ";
235          }
236          cout<<"\n";
237      }
238
239      cout<<"Matrix B :\n";
240      for(int i = 0; i<ROW_SIZE; ++i){
241          for(int j = 0; j<COL_SIZE; ++j){
242              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
243          }
244          cout<<"\n";
245      }
246
247      cout<<"Matrix Transposed B :\n";
248      for(int i = 0; i<ROW_SIZE; ++i){
249          for(int j = 0; j<COL_SIZE; ++j){
250              cout<<mat_b[i*COL_SIZE+j]<<" ";
251          }
252          cout<<"\n";
253      }
254
255      cout<<"Matrix Result :\n";
256      for(int i = 0; i<ROW_SIZE; ++i){
257          for(int j = 0; j<COL_SIZE; ++j){
258              cout<<res[i*COL_SIZE+j]<<" ";
259          }
260          cout<<"\n";
261      }
262
263      cout<<"\n";
264
265      cout<<"Matrix A :\n";
266      for(int i = 0; i<ROW_SIZE; ++i){
267          for(int j = 0; j<COL_SIZE; ++j){
268              cout<<mat_a[i*COL_SIZE+j]<<" ";
269          }
270          cout<<"\n";
271      }
272
273      cout<<"Matrix B :\n";
274      for(int i = 0; i<ROW_SIZE; ++i){
275          for(int j = 0; j<COL_SIZE; ++j){
276              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
277          }
278          cout<<"\n";
279      }
280
281      cout<<"Matrix Transposed B :\n";
282      for(int i = 0; i<ROW_SIZE; ++i){
283          for(int j = 0; j<COL_SIZE; ++j){
284              cout<<mat_b[i*COL_SIZE+j]<<" ";
285          }
286          cout<<"\n";
287      }
288
289      cout<<"Matrix Result :\n";
290      for(int i = 0; i<ROW_SIZE; ++i){
291          for(int j = 0; j<COL_SIZE; ++j){
292              cout<<res[i*COL_SIZE+j]<<" ";
293          }
294          cout<<"\n";
295      }
296
297      cout<<"\n";
298
299      cout<<"Matrix A :\n";
300      for(int i = 0; i<ROW_SIZE; ++i){
301          for(int j = 0; j<COL_SIZE; ++j){
302              cout<<mat_a[i*COL_SIZE+j]<<" ";
303          }
304          cout<<"\n";
305      }
306
307      cout<<"Matrix B :\n";
308      for(int i = 0; i<ROW_SIZE; ++i){
309          for(int j = 0; j<COL_SIZE; ++j){
310              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
311          }
312          cout<<"\n";
313      }
314
315      cout<<"Matrix Transposed B :\n";
316      for(int i = 0; i<ROW_SIZE; ++i){
317          for(int j = 0; j<COL_SIZE; ++j){
318              cout<<mat_b[i*COL_SIZE+j]<<" ";
319          }
320          cout<<"\n";
321      }
322
323      cout<<"Matrix Result :\n";
324      for(int i = 0; i<ROW_SIZE; ++i){
325          for(int j = 0; j<COL_SIZE; ++j){
326              cout<<res[i*COL_SIZE+j]<<" ";
327          }
328          cout<<"\n";
329      }
330
331      cout<<"\n";
332
333      cout<<"Matrix A :\n";
334      for(int i = 0; i<ROW_SIZE; ++i){
335          for(int j = 0; j<COL_SIZE; ++j){
336              cout<<mat_a[i*COL_SIZE+j]<<" ";
337          }
338          cout<<"\n";
339      }
340
341      cout<<"Matrix B :\n";
342      for(int i = 0; i<ROW_SIZE; ++i){
343          for(int j = 0; j<COL_SIZE; ++j){
344              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
345          }
346          cout<<"\n";
347      }
348
349      cout<<"Matrix Transposed B :\n";
350      for(int i = 0; i<ROW_SIZE; ++i){
351          for(int j = 0; j<COL_SIZE; ++j){
352              cout<<mat_b[i*COL_SIZE+j]<<" ";
353          }
354          cout<<"\n";
355      }
356
357      cout<<"Matrix Result :\n";
358      for(int i = 0; i<ROW_SIZE; ++i){
359          for(int j = 0; j<COL_SIZE; ++j){
360              cout<<res[i*COL_SIZE+j]<<" ";
361          }
362          cout<<"\n";
363      }
364
365      cout<<"\n";
366
367      cout<<"Matrix A :\n";
368      for(int i = 0; i<ROW_SIZE; ++i){
369          for(int j = 0; j<COL_SIZE; ++j){
370              cout<<mat_a[i*COL_SIZE+j]<<" ";
371          }
372          cout<<"\n";
373      }
374
375      cout<<"Matrix B :\n";
376      for(int i = 0; i<ROW_SIZE; ++i){
377          for(int j = 0; j<COL_SIZE; ++j){
378              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
379          }
380          cout<<"\n";
381      }
382
383      cout<<"Matrix Transposed B :\n";
384      for(int i = 0; i<ROW_SIZE; ++i){
385          for(int j = 0; j<COL_SIZE; ++j){
386              cout<<mat_b[i*COL_SIZE+j]<<" ";
387          }
388          cout<<"\n";
389      }
390
391      cout<<"Matrix Result :\n";
392      for(int i = 0; i<ROW_SIZE; ++i){
393          for(int j = 0; j<COL_SIZE; ++j){
394              cout<<res[i*COL_SIZE+j]<<" ";
395          }
396          cout<<"\n";
397      }
398
399      cout<<"\n";
400
401      cout<<"Matrix A :\n";
402      for(int i = 0; i<ROW_SIZE; ++i){
403          for(int j = 0; j<COL_SIZE; ++j){
404              cout<<mat_a[i*COL_SIZE+j]<<" ";
405          }
406          cout<<"\n";
407      }
408
409      cout<<"Matrix B :\n";
410      for(int i = 0; i<ROW_SIZE; ++i){
411          for(int j = 0; j<COL_SIZE; ++j){
412              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
413          }
414          cout<<"\n";
415      }
416
417      cout<<"Matrix Transposed B :\n";
418      for(int i = 0; i<ROW_SIZE; ++i){
419          for(int j = 0; j<COL_SIZE; ++j){
420              cout<<mat_b[i*COL_SIZE+j]<<" ";
421          }
422          cout<<"\n";
423      }
424
425      cout<<"Matrix Result :\n";
426      for(int i = 0; i<ROW_SIZE; ++i){
427          for(int j = 0; j<COL_SIZE; ++j){
428              cout<<res[i*COL_SIZE+j]<<" ";
429          }
430          cout<<"\n";
431      }
432
433      cout<<"\n";
434
435      cout<<"Matrix A :\n";
436      for(int i = 0; i<ROW_SIZE; ++i){
437          for(int j = 0; j<COL_SIZE; ++j){
438              cout<<mat_a[i*COL_SIZE+j]<<" ";
439          }
440          cout<<"\n";
441      }
442
443      cout<<"Matrix B :\n";
444      for(int i = 0; i<ROW_SIZE; ++i){
445          for(int j = 0; j<COL_SIZE; ++j){
446              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
447          }
448          cout<<"\n";
449      }
450
451      cout<<"Matrix Transposed B :\n";
452      for(int i = 0; i<ROW_SIZE; ++i){
453          for(int j = 0; j<COL_SIZE; ++j){
454              cout<<mat_b[i*COL_SIZE+j]<<" ";
455          }
456          cout<<"\n";
457      }
458
459      cout<<"Matrix Result :\n";
460      for(int i = 0; i<ROW_SIZE; ++i){
461          for(int j = 0; j<COL_SIZE; ++j){
462              cout<<res[i*COL_SIZE+j]<<" ";
463          }
464          cout<<"\n";
465      }
466
467      cout<<"\n";
468
469      cout<<"Matrix A :\n";
470      for(int i = 0; i<ROW_SIZE; ++i){
471          for(int j = 0; j<COL_SIZE; ++j){
472              cout<<mat_a[i*COL_SIZE+j]<<" ";
473          }
474          cout<<"\n";
475      }
476
477      cout<<"Matrix B :\n";
478      for(int i = 0; i<ROW_SIZE; ++i){
479          for(int j = 0; j<COL_SIZE; ++j){
480              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
481          }
482          cout<<"\n";
483      }
484
485      cout<<"Matrix Transposed B :\n";
486      for(int i = 0; i<ROW_SIZE; ++i){
487          for(int j = 0; j<COL_SIZE; ++j){
488              cout<<mat_b[i*COL_SIZE+j]<<" ";
489          }
490          cout<<"\n";
491      }
492
493      cout<<"Matrix Result :\n";
494      for(int i = 0; i<ROW_SIZE; ++i){
495          for(int j = 0; j<COL_SIZE; ++j){
496              cout<<res[i*COL_SIZE+j]<<" ";
497          }
498          cout<<"\n";
499      }
500
501      cout<<"\n";
502
503      cout<<"Matrix A :\n";
504      for(int i = 0; i<ROW_SIZE; ++i){
505          for(int j = 0; j<COL_SIZE; ++j){
506              cout<<mat_a[i*COL_SIZE+j]<<" ";
507          }
508          cout<<"\n";
509      }
510
511      cout<<"Matrix B :\n";
512      for(int i = 0; i<ROW_SIZE; ++i){
513          for(int j = 0; j<COL_SIZE; ++j){
514              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
515          }
516          cout<<"\n";
517      }
518
519      cout<<"Matrix Transposed B :\n";
520      for(int i = 0; i<ROW_SIZE; ++i){
521          for(int j = 0; j<COL_SIZE; ++j){
522              cout<<mat_b[i*COL_SIZE+j]<<" ";
523          }
524          cout<<"\n";
525      }
526
527      cout<<"Matrix Result :\n";
528      for(int i = 0; i<ROW_SIZE; ++i){
529          for(int j = 0; j<COL_SIZE; ++j){
530              cout<<res[i*COL_SIZE+j]<<" ";
531          }
532          cout<<"\n";
533      }
534
535      cout<<"\n";
536
537      cout<<"Matrix A :\n";
538      for(int i = 0; i<ROW_SIZE; ++i){
539          for(int j = 0; j<COL_SIZE; ++j){
540              cout<<mat_a[i*COL_SIZE+j]<<" ";
541          }
542          cout<<"\n";
543      }
544
545      cout<<"Matrix B :\n";
546      for(int i = 0; i<ROW_SIZE; ++i){
547          for(int j = 0; j<COL_SIZE; ++j){
548              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
549          }
550          cout<<"\n";
551      }
552
553      cout<<"Matrix Transposed B :\n";
554      for(int i = 0; i<ROW_SIZE; ++i){
555          for(int j = 0; j<COL_SIZE; ++j){
556              cout<<mat_b[i*COL_SIZE+j]<<" ";
557          }
558          cout<<"\n";
559      }
560
561      cout<<"Matrix Result :\n";
562      for(int i = 0; i<ROW_SIZE; ++i){
563          for(int j = 0; j<COL_SIZE; ++j){
564              cout<<res[i*COL_SIZE+j]<<" ";
565          }
566          cout<<"\n";
567      }
568
569      cout<<"\n";
570
571      cout<<"Matrix A :\n";
572      for(int i = 0; i<ROW_SIZE; ++i){
573          for(int j = 0; j<COL_SIZE; ++j){
574              cout<<mat_a[i*COL_SIZE+j]<<" ";
575          }
576          cout<<"\n";
577      }
578
579      cout<<"Matrix B :\n";
580      for(int i = 0; i<ROW_SIZE; ++i){
581          for(int j = 0; j<COL_SIZE; ++j){
582              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
583          }
584          cout<<"\n";
585      }
586
587      cout<<"Matrix Transposed B :\n";
588      for(int i = 0; i<ROW_SIZE; ++i){
589          for(int j = 0; j<COL_SIZE; ++j){
590              cout<<mat_b[i*COL_SIZE+j]<<" ";
591          }
592          cout<<"\n";
593      }
594
595      cout<<"Matrix Result :\n";
596      for(int i = 0; i<ROW_SIZE; ++i){
597          for(int j = 0; j<COL_SIZE; ++j){
598              cout<<res[i*COL_SIZE+j]<<" ";
599          }
600          cout<<"\n";
601      }
602
603      cout<<"\n";
604
605      cout<<"Matrix A :\n";
606      for(int i = 0; i<ROW_SIZE; ++i){
607          for(int j = 0; j<COL_SIZE; ++j){
608              cout<<mat_a[i*COL_SIZE+j]<<" ";
609          }
610          cout<<"\n";
611      }
612
613      cout<<"Matrix B :\n";
614      for(int i = 0; i<ROW_SIZE; ++i){
615          for(int j = 0; j<COL_SIZE; ++j){
616              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
617          }
618          cout<<"\n";
619      }
620
621      cout<<"Matrix Transposed B :\n";
622      for(int i = 0; i<ROW_SIZE; ++i){
623          for(int j = 0; j<COL_SIZE; ++j){
624              cout<<mat_b[i*COL_SIZE+j]<<" ";
625          }
626          cout<<"\n";
627      }
628
629      cout<<"Matrix Result :\n";
630      for(int i = 0; i<ROW_SIZE; ++i){
631          for(int j = 0; j<COL_SIZE; ++j){
632              cout<<res[i*COL_SIZE+j]<<" ";
633          }
634          cout<<"\n";
635      }
636
637      cout<<"\n";
638
639      cout<<"Matrix A :\n";
640      for(int i = 0; i<ROW_SIZE; ++i){
641          for(int j = 0; j<COL_SIZE; ++j){
642              cout<<mat_a[i*COL_SIZE+j]<<" ";
643          }
644          cout<<"\n";
645      }
646
647      cout<<"Matrix B :\n";
648      for(int i = 0; i<ROW_SIZE; ++i){
649          for(int j = 0; j<COL_SIZE; ++j){
650              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
651          }
652          cout<<"\n";
653      }
654
655      cout<<"Matrix Transposed B :\n";
656      for(int i = 0; i<ROW_SIZE; ++i){
657          for(int j = 0; j<COL_SIZE; ++j){
658              cout<<mat_b[i*COL_SIZE+j]<<" ";
659          }
660          cout<<"\n";
661      }
662
663      cout<<"Matrix Result :\n";
664      for(int i = 0; i<ROW_SIZE; ++i){
665          for(int j = 0; j<COL_SIZE; ++j){
666              cout<<res[i*COL_SIZE+j]<<" ";
667          }
668          cout<<"\n";
669      }
670
671      cout<<"\n";
672
673      cout<<"Matrix A :\n";
674      for(int i = 0; i<ROW_SIZE; ++i){
675          for(int j = 0; j<COL_SIZE; ++j){
676              cout<<mat_a[i*COL_SIZE+j]<<" ";
677          }
678          cout<<"\n";
679      }
680
681      cout<<"Matrix B :\n";
682      for(int i = 0; i<ROW_SIZE; ++i){
683          for(int j = 0; j<COL_SIZE; ++j){
684              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
685          }
686          cout<<"\n";
687      }
688
689      cout<<"Matrix Transposed B :\n";
690      for(int i = 0; i<ROW_SIZE; ++i){
691          for(int j = 0; j<COL_SIZE; ++j){
692              cout<<mat_b[i*COL_SIZE+j]<<" ";
693          }
694          cout<<"\n";
695      }
696
697      cout<<"Matrix Result :\n";
698      for(int i = 0; i<ROW_SIZE; ++i){
699          for(int j = 0; j<COL_SIZE; ++j){
700              cout<<res[i*COL_SIZE+j]<<" ";
701          }
702          cout<<"\n";
703      }
704
705      cout<<"\n";
706
707      cout<<"Matrix A :\n";
708      for(int i = 0; i<ROW_SIZE; ++i){
709          for(int j = 0; j<COL_SIZE; ++j){
710              cout<<mat_a[i*COL_SIZE+j]<<" ";
711          }
712          cout<<"\n";
713      }
714
715      cout<<"Matrix B :\n";
716      for(int i = 0; i<ROW_SIZE; ++i){
717          for(int j = 0; j<COL_SIZE; ++j){
718              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
719          }
720          cout<<"\n";
721      }
722
723      cout<<"Matrix Transposed B :\n";
724      for(int i = 0; i<ROW_SIZE; ++i){
725          for(int j = 0; j<COL_SIZE; ++j){
726              cout<<mat_b[i*COL_SIZE+j]<<" ";
727          }
728          cout<<"\n";
729      }
730
731      cout<<"Matrix Result :\n";
732      for(int i = 0; i<ROW_SIZE; ++i){
733          for(int j = 0; j<COL_SIZE; ++j){
734              cout<<res[i*COL_SIZE+j]<<" ";
735          }
736          cout<<"\n";
737      }
738
739      cout<<"\n";
740
741      cout<<"Matrix A :\n";
742      for(int i = 0; i<ROW_SIZE; ++i){
743          for(int j = 0; j<COL_SIZE; ++j){
744              cout<<mat_a[i*COL_SIZE+j]<<" ";
745          }
746          cout<<"\n";
747      }
748
749      cout<<"Matrix B :\n";
750      for(int i = 0; i<ROW_SIZE; ++i){
751          for(int j = 0; j<COL_SIZE; ++j){
752              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
753          }
754          cout<<"\n";
755      }
756
757      cout<<"Matrix Transposed B :\n";
758      for(int i = 0; i<ROW_SIZE; ++i){
759          for(int j = 0; j<COL_SIZE; ++j){
760              cout<<mat_b[i*COL_SIZE+j]<<" ";
761          }
762          cout<<"\n";
763      }
764
765      cout<<"Matrix Result :\n";
766      for(int i = 0; i<ROW_SIZE; ++i){
767          for(int j = 0; j<COL_SIZE; ++j){
768              cout<<res[i*COL_SIZE+j]<<" ";
769          }
770          cout<<"\n";
771      }
772
773      cout<<"\n";
774
775      cout<<"Matrix A :\n";
776      for(int i = 0; i<ROW_SIZE; ++i){
777          for(int j = 0; j<COL_SIZE; ++j){
778              cout<<mat_a[i*COL_SIZE+j]<<" ";
779          }
780          cout<<"\n";
781      }
782
783      cout<<"Matrix B :\n";
784      for(int i = 0; i<ROW_SIZE; ++i){
785          for(int j = 0; j<COL_SIZE; ++j){
786              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
787          }
788          cout<<"\n";
789      }
790
791      cout<<"Matrix Transposed B :\n";
792      for(int i = 0; i<ROW_SIZE; ++i){
793          for(int j = 0; j<COL_SIZE; ++j){
794              cout<<mat_b[i*COL_SIZE+j]<<" ";
795          }
796          cout<<"\n";
797      }
798
799      cout<<"Matrix Result :\n";
800      for(int i = 0; i<ROW_SIZE; ++i){
801          for(int j = 0; j<COL_SIZE; ++j){
802              cout<<res[i*COL_SIZE+j]<<" ";
803          }
804          cout<<"\n";
805      }
806
807      cout<<"\n";
808
809      cout<<"Matrix A :\n";
810      for(int i = 0; i<ROW_SIZE; ++i){
811          for(int j = 0; j<COL_SIZE; ++j){
812              cout<<mat_a[i*COL_SIZE+j]<<" ";
813          }
814          cout<<"\n";
815      }
816
817      cout<<"Matrix B :\n";
818      for(int i = 0; i<ROW_SIZE; ++i){
819          for(int j = 0; j<COL_SIZE; ++j){
820              cout<<mat_b_T[i*COL_SIZE+j]<<" ";
821          }
822          cout<<"\n";
823      }
824
825      cout<&
```

```

33     for(int i = 0;i<ROW_SIZE;++i){
34         for(int j = 0;j<COL_SIZE;++j){
35             double sum{0.0};
36             for(int k = 0;k<ROW_SIZE;++k){
37                 sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
38             }
39             res[i*COL_SIZE+j] = sum;
40         }
41     }
42     auto end_time{chrono::steady_clock::now()};
43     auto exec_time{end_time-start_time};
44     chrono::duration<double, milli> ms{exec_time};
45     cout<<"Execution time:"<<ms.count()<<endl;
46
47     cout<<"Using 1D Threading"<<endl;
48     start_time = chrono::steady_clock::now();
49     #pragma omp parallel for
50     for(int i = 0;i<ROW_SIZE;++i){
51         for(int j = 0;j<COL_SIZE;++j){
52             double sum{0.0};
53             for(int k = 0;k<ROW_SIZE;++k){
54                 sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
55             }
56             res[i*COL_SIZE+j] = sum;
57         }
58     }
59     end_time = chrono::steady_clock::now();
60     exec_time = end_time-start_time;
61     ms = exec_time;
62     cout<<"Execution time: "<< ms.count()<<"\n";
63
64     cout<<"Using 2D Threading"<<endl;
65     start_time = chrono::steady_clock::now();
66     #pragma omp parallel for
67     for(int i = 0;i<ROW_SIZE;++i){
68         #pragma omp parallel for
69         for(int j = 0;j<COL_SIZE;++j){
70             double sum{0.0};
71             for(int k = 0;k<ROW_SIZE;++k){
72                 sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
73             }
74             res[i*COL_SIZE+j] = sum;
75         }
76     }

```

```

3     end_time = chrono::steady_clock::now();
4     exec_time = end_time-start_time;
5     ms = exec_time;
6     cout<<"Execution time:"<<ms.count()<<endl;
7
8     cout<<"Using 2D Threading (collapsed)"<<"\n";
9     start_time = chrono::steady_clock::now();
10    #pragma omp parallel for collapse(2)
11    for(int i = 0;i<ROW_SIZE;++i){
12        for(int j = 0;j<COL_SIZE;++j){
13            double sum{0.0};
14            for(int k = 0;k<ROW_SIZE;++k){
15                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
16            }
17            res[i*COL_SIZE+j] = sum;
18        }
19    }
20    end_time = chrono::steady_clock::now();
21    exec_time = end_time-start_time;
22    ms = exec_time;
23    cout<< "Execution time: "<<ms.count()<<endl;
24    return 0;
25 }
26

```

INSERT ➤ ↵ main > ~3 -2 <1/matrix_multiplication.cpp [+] utf-8 < ▲ < ⌂ cpp ◀ 76% ◀ 83:18

Terminal Output:

```
▲ all_might ~ >> cd ~/Documents/UCS645/LAB1
▲ all_might ~/Documents/UCS645/LAB1 git-[* main]- >> perf stat ./matrix_multiplication
Using No Threading
Execution time: 1485.69
Using 1D Threading
Execution time: 117.708
Using 2D Threading
Execution time: 119.167
Using 2D Threading (collapsed)

Performance counter stats for './matrix_multiplication':

      6,607,021,593    task-clock:u          #  3.449 CPUs utilized
                  0    context-switches:u      #  0.000 /sec
                  0    cpu-migrations:u        #  0.000 /sec
                 8,031    page-faults:u         #  1.216 K/sec
     18,422,366,808    instructions:u       #  1.01  insn per cycle
     18,319,571,404    cycles:u            #  2.773 GHz
     2,052,631,886    branches:u          # 310.674 M/sec
        4,132,451    branch-misses:u       #  0.20% of all branches

  1.915598446 seconds time elapsed

  6.576039000 seconds user
  0.031995000 seconds sys
```

Performance Results :

Strategy	Execution Time (ms)	Speedup Factor	Analysis
Serial (No Threading)	1485.69 ms	1.0x (Baseline)	The Control Group. This baseline represents the worst case scenario where a single core must process all 1,000,000 iterations sequentially. The high time reflects the sheer volume of floating point operations required N^3 complexity.
Parallel (1D Threading)	117.71 ms	12.62x	Top Performer. By parallelizing the outer loop, we effectively split the 1,000 rows among the available threads. The chunk size work per thread was large enough to keep every core busy without frequent synchronization, leading to the best performance.
Parallel (2D Threading)	119.17 ms	12.46x	Diminishing Returns. While still vastly faster than serial, it is slightly slower than 1D threading. Creating threads for both the inner and outer loops added unnecessary management overhead that slightly outweighed the benefits of finer grained parallelism.

Architectural Analysis :

- **Observation:** The speedup (approximately 12.6x) exceeds the physical core count of many standard laptops, suggesting that Hyper threading and efficient pipeline utilization (SIMD instructions) played a role.
- **perf stat** Insight:

Instructions per Cycle IPC: 1.01. This is a healthy IPC for a mixed workload, indicating the CPU pipeline was rarely stalled waiting for data, proving the Transposition optimization worked.

Branch Misses : - 0.20% which is Extremely low. The predictable loop structure allowed the CPU's **Branch Predictor** to be highly accurate.

Experiment 3: Monte Carlo Pi Calculation

Analyzing a "Compute Bound" workload and Reduction clauses.

Methodology & Critical Concepts

This algorithm estimates pi by summing the area of rectangles under a curve. It requires almost no memory access but heavy arithmetic operations.

- **The Challenge:** A "Race Condition." If all threads add to a single global sum variable simultaneously, data corruption occurs.
- **The Solution:** OpenMP Reduction.

Code :

```
matrix_multiplication.cpp          pi.cpp
1 #include <chrono>
1 #include <print>
2 #include <ratio>
3 #include<iostream>
4 using namespace std;
5 int main(){
6     int num_steps{static_cast<int>(1e9)};
7     double pi{};
8     double sum{0.0};
9     double step{1.0/num_steps};
10
11    cout<<"With No Threading"<<endl;
12    auto start_time{ chrono::steady_clock::now()};
13    for(int i = 0;i<num_steps;++i){
14        double x = (i+0.5)*step;
15        sum += 4.0/(1.0+x*x);
16    }
17    pi = step*sum;
18    auto end_time{ chrono::steady_clock::now()};
19    auto exec_time{end_time-start_time};
20    chrono::duration<double> milli{ms(exec_time)};
21    cout<<"Execution Time: "<<ms.count()<<endl;
22    cout<< "Pi (Serial): "<<pi<<endl;
23
24    sum = 0.0;
25    cout<<"With Threading"<<endl;
26    start_time = chrono::steady_clock::now();
27    #pragma omp parallel for reduction(+:sum)
28    for(int i = 0;i<num_steps;++i){
29        double x = (i+0.5)*step;
30        sum += 4.0/(1.0+x*x);
31    }
32    pi = step*sum;
33    end_time = chrono::steady_clock::now();
34    exec_time = end_time-start_time;
35    ms = exec_time;
36    cout<<"Execution Time: "<<ms.count()<<endl;
37    cout<<"Pi (Parallel): "<<pi<<endl;
38
39 }
```

NORMAL ➤ **main** ➤ ? ➤ Documents/UCS645/LAB1/pi.cpp utf-8 < Δ < @cpp Top 1:1
E21: Cannot make changes, 'modifiable' is off

#pragma omp parallel for reduction(+:sum)

The reduction clause creates a **private copy** of the variable sum for each thread in its local stack. Threads compute independently without locking. At the end of the parallel region, the master thread reduces all private copies into the global result. This eliminates False Sharing and cache contention.

```
A all_might ✘ ~/Documents/UCS645/LAB1 git-[v main]- >> perf stat ./pi
With No Threading
Execution Time: 1130.38
Pi (Serial): 3.14159
With Threading
Execution Time: 107.078
Pi (Parallel): 3.14159

Performance counter stats for './pi':

      2,443,196,204      task-clock:u          #      1.921 CPUs utilized
                  0      context-switches:u        #      0.000 /sec
                  0      cpu-migrations:u        #      0.000 /sec
                 174      page-faults:u          #    71.218 /sec
      13,508,971,914      instructions:u        #      1.45  insn per cycle
      9,333,810,041      cycles:u            #      3.820 GHz
      502,365,221       branches:u          #   205.618 M/sec
                 12,365      branch-misses:u        #      0.00% of all branches

      1.271703881 seconds time elapsed

      2.440258000 seconds user
      0.003993000 seconds sys
```

Performance Results :

Strategy	Execution Time (ms)	Speedup Factor	Teacher's Analysis
Serial (No Threading)	1130.38 ms	1.0x Baseline	Single Lane Traffic. The CPU executes the loop one iteration at a time. Since the calculation $\frac{4}{1+x^2}$ is purely mathematical, the CPU is effectively "speed limited" by the clock speed of a single core.
Parallel (Reduction)	107.08 ms	10.55x	Near Linear Scaling. Because there is almost no memory traffic to slow things down, the performance scales directly with the number of cores. The reduction clause allowed every core to work at full speed without waiting for others, achieving a massive 10x improvement.

Architectural Analysis :

- **Why 10x Speedup:**

Since the task is compute bound, it scales almost linearly with the number of available threads. There is no bottleneck from RAM speed.

- **perf stat Insight:**

IPC: 1.45. This is significantly higher than the Matrix experiment. Since the CPU is doing pure math (floating point ops) inside registers and rarely accessing RAM, the pipeline runs at maximum efficiency.

Inferences

Achieving maximum efficiency in high performance computing is not about maximizing thread count, but rather about optimizing the **ratio of useful computation to management overhead**.

Our experiments reveal that parallel scalability is governed by two opposing forces: **Granularity** and **Resource Contention**.

1. The Granularity Threshold:

In the **Matrix Multiplication** experiment, the workload was **Coarse Grained** heavy computation per thread. The CPU spent the vast majority of its cycles executing floating point operations, making the initial cost of creating threads negligible. This resulted in a massive **12.6x speedup**.

In contrast, the **DAXPY** experiment represented a **Fine Grained** workload. With a small dataset, the loop body executed so quickly that the **overhead** of spawning, scheduling, and synchronizing 12 threads outweighed the time saved by parallelizing the math. This explains why performance degraded at 12 threads i.e 6.92 ms compared to 11 threads i.e 2.98 ms.

2. Hardware Saturation :

Parallelism is limited by the slowest component in the system. The **DAXPY perf stat** revealed an extremely low **IPC of 0.24**, proving the CPU cores were starving for data. This indicates that we hit the **Memory Bandwidth Wall**—adding more threads could not process data faster because the physical link between the CPU and RAM was fully saturated.

Therefore, efficient parallel architecture requires a "Sweet Spot" approach. We must ensure the **Parallel Region** is computationally intense enough to justify the overhead of threading, while respecting the physical limits of the memory subsystem. Simply adding cores to a memory bound or lightweight task will result in **diminishing returns** and eventually **performance degradation**.

Conclusion

This set of experiments successfully demonstrated three key principles of parallel architecture:

1. **Optimization First:** Parallelism cannot fix bad code. The Matrix Transposition Algorithmic change was just as vital as the threading itself.
2. **Workload Dependencies:** Compute bound tasks Pi scale perfectly. Memory bound tasks DAXPY hit a "wall" where adding threads degrades performance.
3. **Synchronization Costs:** Efficient parallel code must use techniques like reduction to avoid the heavy cost of locks and critical sections.