Parallel Implementations of LU Decomposition

Hardik Khichi
2016CS50404

Saransh Verma
2016CS10326

24th January, 2020

COL380: Introduction to Parallel and Distributed Programming - Assignment 1

1 Algorithm Overview

We want to develop program for LU decomposition that used Gaussian elimination to factor a dense N x N matrix into an upper-triangular one and a lower-triangular one. In matrix computations, pivoting involves finding the largest magnitude value in a row, column, or both and then interchanging rows and/or columns in the matrix for the next step in the algorithm. The purpose of pivoting is to reduce round-off error, which enhances numerical stability. In our programs, we will use row pivoting, a form of pivoting involves interchanging rows of a trailing sub-matrix based on the largest value in the current column. To perform LU decomposition with row pivoting, we will compute a permutation matrix P such that PA = LU. The permutation matrix keeps track of row exchanges performed.

To check our answer, we compute the sum of Euclidean norms of the columns of the residual matrix computed as PA-LU. We use the value of the L1 norm of the residual. (It comes out to be very small.)

Each LU decomposition implementation accepts two arguments: n - the size of a matrix, followed by t - the number of threads.

1.1 Sequential Program:

- Initialize a as a n x n matrix of double precision (64-bit) floating point random values generated using rand(). We keep seed for random value generation as constant, srand(1). This is done so that we generate the same matrix all implementations.
- Initialize π as a array of length n with all values 0 to n.
- Initialize u as an n x n matrix with all 0s double values.
- Initialize l as an n x n matrix with all 0s double values expect 1s on the diagonal.
- For each iteration k from 0 to n-1:
  - We select the index k' which contains the largest value in upper triangle of a.
  - If max value while calculating k' comes out to be 0. Raise error: Simgular Matrix.
  - Swap the following row values in a, l and π (for row pivoting):
    - π[k] with π[k']

    * a[k][ ] with a[k’][ ]
    * l[k][0:k-1] with l[k’][0:k-1]
  - Put u[k][k] = a[k][k]
  - for i = k+1 to n
    * l(i,k) = a(i,k)/u(k,k)
    * u(k,i) = a(k,i)
  - for i = k+1 to n
    * for j = k+1 to n
    * a(i,j) = a(i,j) - l(i,k)*u(k,j)

Pseudo-Code for sequential program:

```
inputs: a(n,n)
outputs:  (n), l(n,n), and u(n,n)

initialize   as a vector of length n
initialize u as an n x n matrix with 0s below the diagonal
initialize l as an n x n matrix with 1s on the diagonal and 0s above the diagonal
for i = 1 to n
    [i] = i
for k = 1 to n
  max = 0
  for i = k to n
    if max < |a(i,k)|
      max = |a(i,k)|
      k’ = i
  if max == 0
    error (singular matrix)
  swap  [k] and  [k’]
  swap a(k,:) and a(k’,:)
  swap l(k,1:k-1) and l(k’,1:k-1)
  u(k,k) = a(k,k)
  for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)
  for i = k+1 to n
    for j = k+1 to n
      a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

1.2 Pthreads Program

:

- We use the sequential program as the base and try to parallelise different components to improve performance.
- We parallelise the last loop in which we subtract l*u elements from a matrix elements. This loop has order O(n 2), hence parallelising this will give us large performance benefits.
- All other loops are in O(n), hence parallelising won’t give us large performance benefits.

Data Structure that stores arguments for threads:

```
struct thread_args{
    int inputSize;
    int k;
    double** a;
    double** l;
    double** u;
    int startPos;
    int endPos;
};
```

Parallel code for nested loop:

```
void* parallelfn(void* argc){
    struct thread_args *t_args = (struct thread_args*) argc;

    for(int i = t_args->startPos+1 ; i < t_args->endPos ; i+=1){
        for(int j = t_args->startPos+1 ; j < n ; j+=1){
            t_args->a[i][j] = t_args->a[i][j] -
                t_args->l[i][t_args->k]*t_args->u[t_args->k][j];
        }
    }

    pthread_exit(0);
}
```

Pthreads implementation:

```
int rowsPerThreads = (n-k)/t;
pthread_t tids[t];
for (int thread = 0; thread < t; thread++)
{
    struct thread_args *targs = (struct thread_args *)malloc(sizeof(struct thread_args));
    targs->inputSize = n;
    targs->a = a;
    targs->l = l;
    targs->u = u;
    targs->k = k;
    targs->startPos = k + thread*rowsPerThreads;
    if(thread != t-1){ targs->endPos = k + (thread+1)*rowsPerThreads; }
    else{ targs->endPos = n; }

    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_create(&tids[thread], &attr, parallelfn, (void*)targs);
}
for(int thread=0; thread < t; thread++){
    pthread_join(tids[thread],NULL);
}
```

1.3 OpenMP Program:

Similar to pthreads we parallelise the nested loop. Code in OpenMP:

```
// Parallelizing the nested loop which has complexity O(n^2) all other steps
// in the loops are linear hence their parallelization will be insignificant
#pragma omp parallel for num_threads(t)
for(int i = k +1 ; i < n ; i +=1){
    for(int j = k + 1 ; j < n ; j +=1){
        a[i][j] = a[i][j] - l[i][k]*u[k][j];
    }
}
#pragma omp barrier
```

2 Design Decisions

- We could have used both C and C++ for this assignment. We have picked C++ in our case because C++ is faster than the former due to its various compiler optimisations. It is also much easier to code in.
- We picked arrays as our choice of data structure instead of vectors because it provides better performance: Execution times for n=1000 for sequential program:
  - Vectors: 27.32 seconds
  - Arrays: 1.50 seconds
- We are only parallelising nested loops with O(n2) complexity as smaller loops won't produce large changes in run-time upon paralellisation

3 Parallelization Strategy

The outer most loop could not be parallelized because of data carried dependencies, inside the outermost loop

4 Load Balancing Strategy

We want that all threads created are given equal work to do. To maintain this we take the following steps for pthread execution:

- We divide number of rows to operate by number of available threads. Work Divide = (k-n)/(t)
- All the

5 Observations

- OpenMP provides better speedup for smaller problems, this may be because in pthread we create new threads in each iteration while OpenMP uses pool of threads.

- Pthreads provide better speedup for larger problems, this may be because it is at a lower level of abstraction, and the cost of creating threads is not very significant in comparison to the cost of the problem.
- For smaller problems execution with smaller number of threads are faster than that with higher number of threads.
- Program runtimes for n=8000 :
  - Sequential = 560.61 seconds
  - OpenMP:

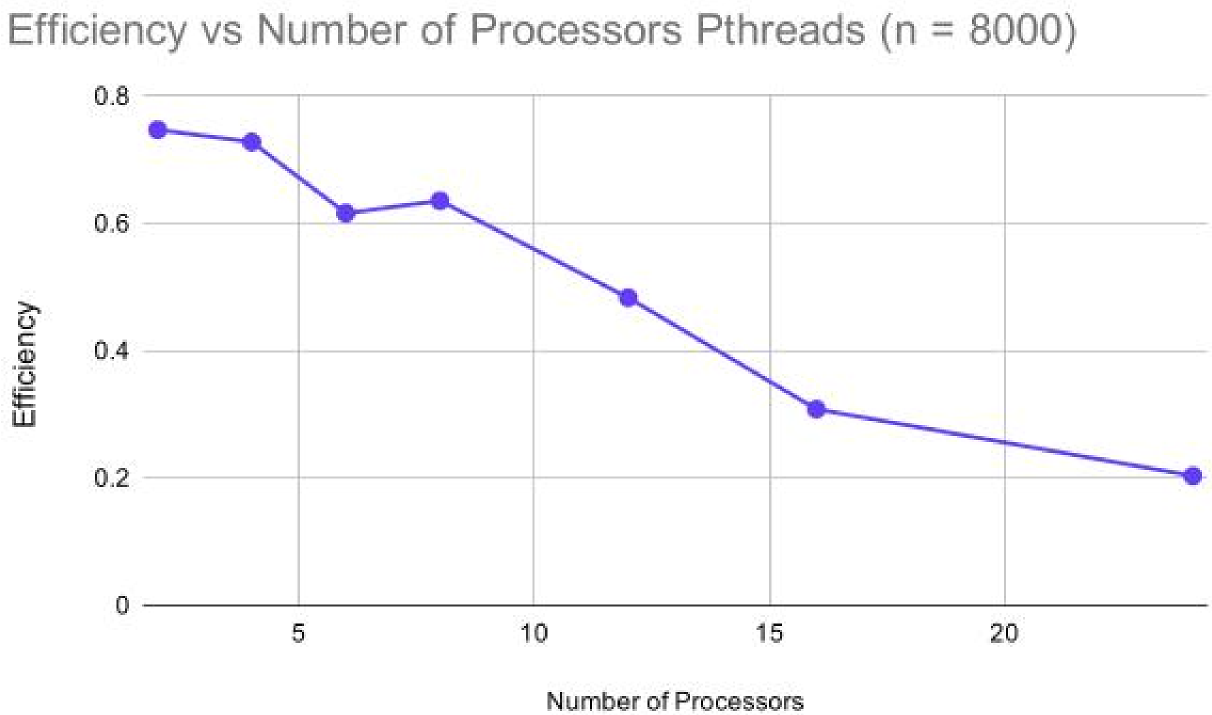
Number of Processors	Time Taken
2	288.61
4	201.17
6	155.8
8	175.76
12	149.95
16	152.17
24	136.92

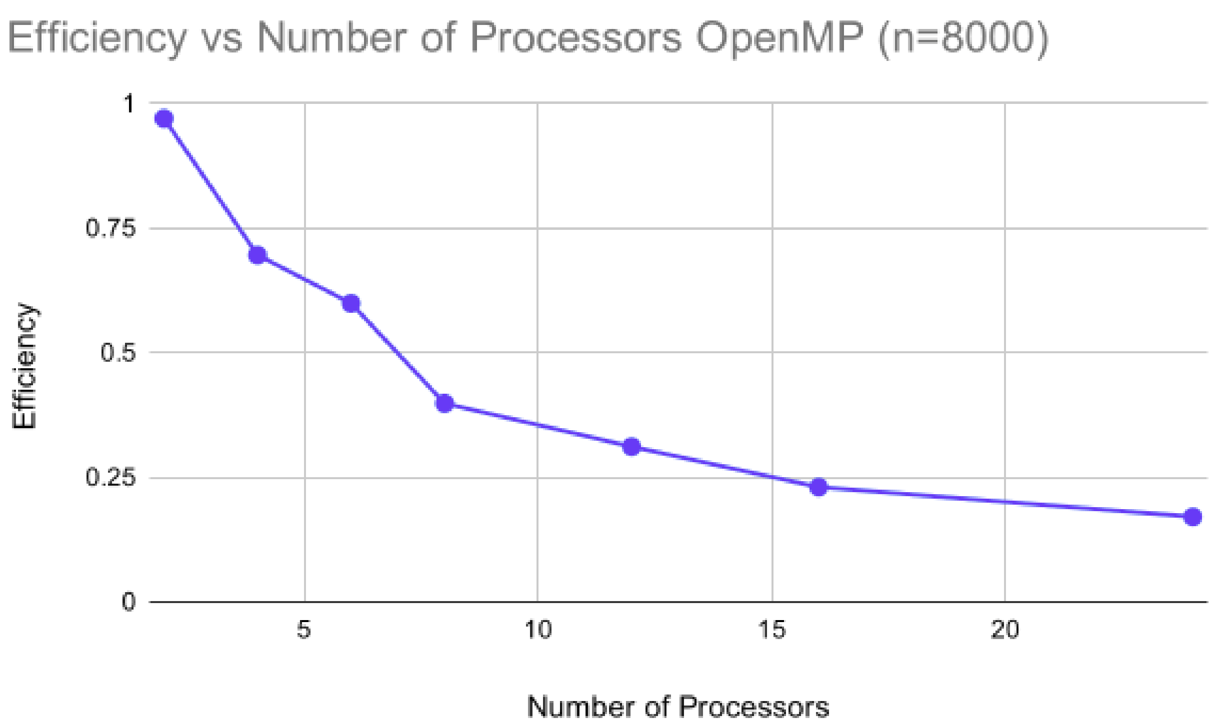
  - Pthreads:

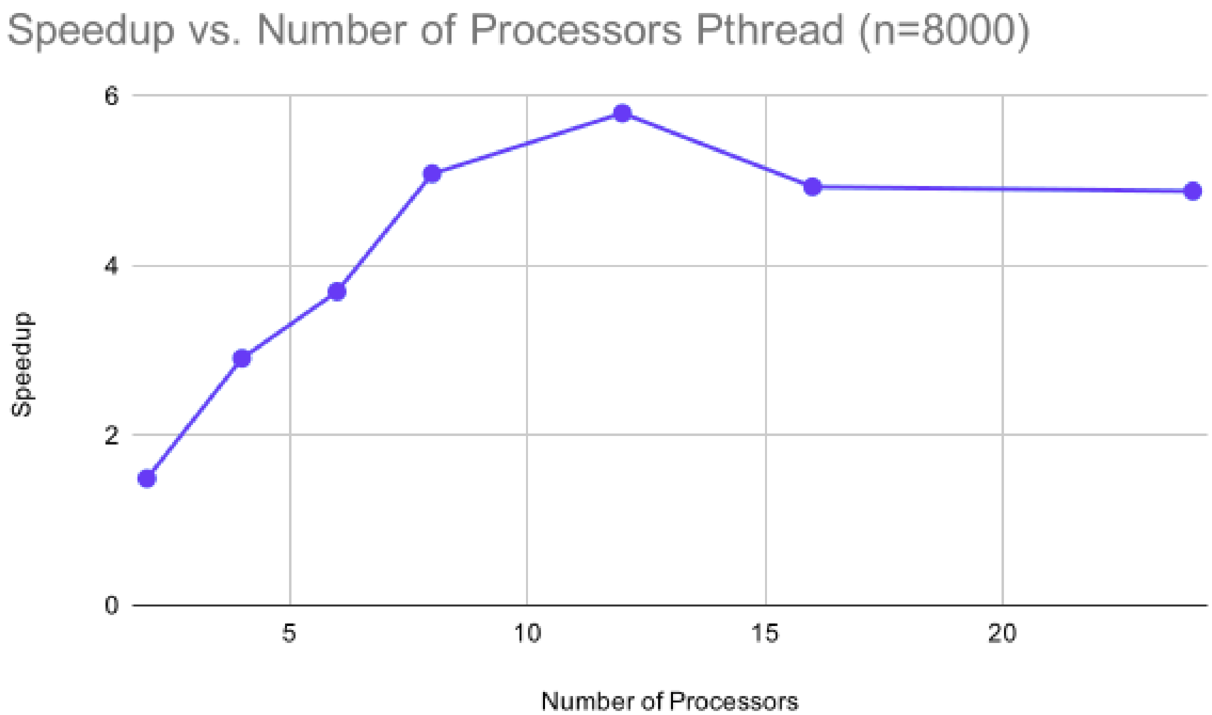
Number of Processors	Time Taken
2	374.74
4	192.32
6	151.47
8	110.12
12	96.56
16	113.57
24	114.73

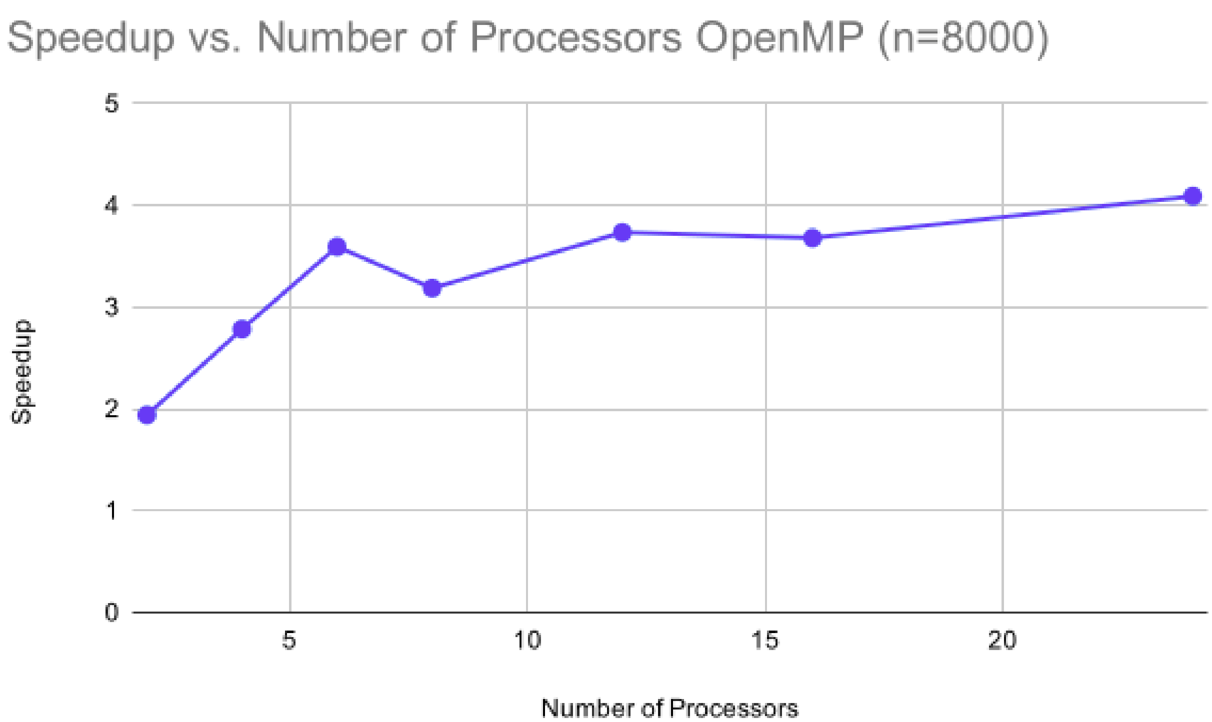
Remarks: Use flag -pthread for compiling pthread execution and -fopenmp for omp execution.

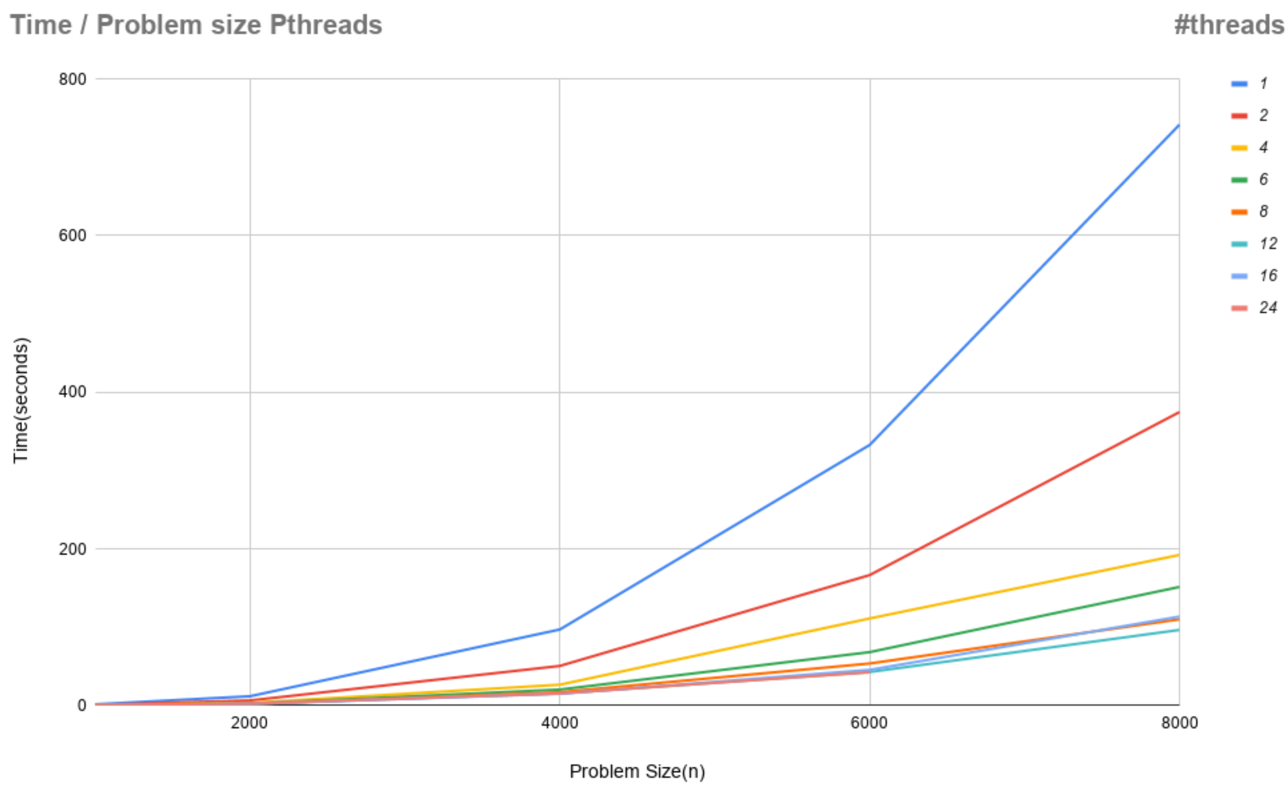
6 Plots

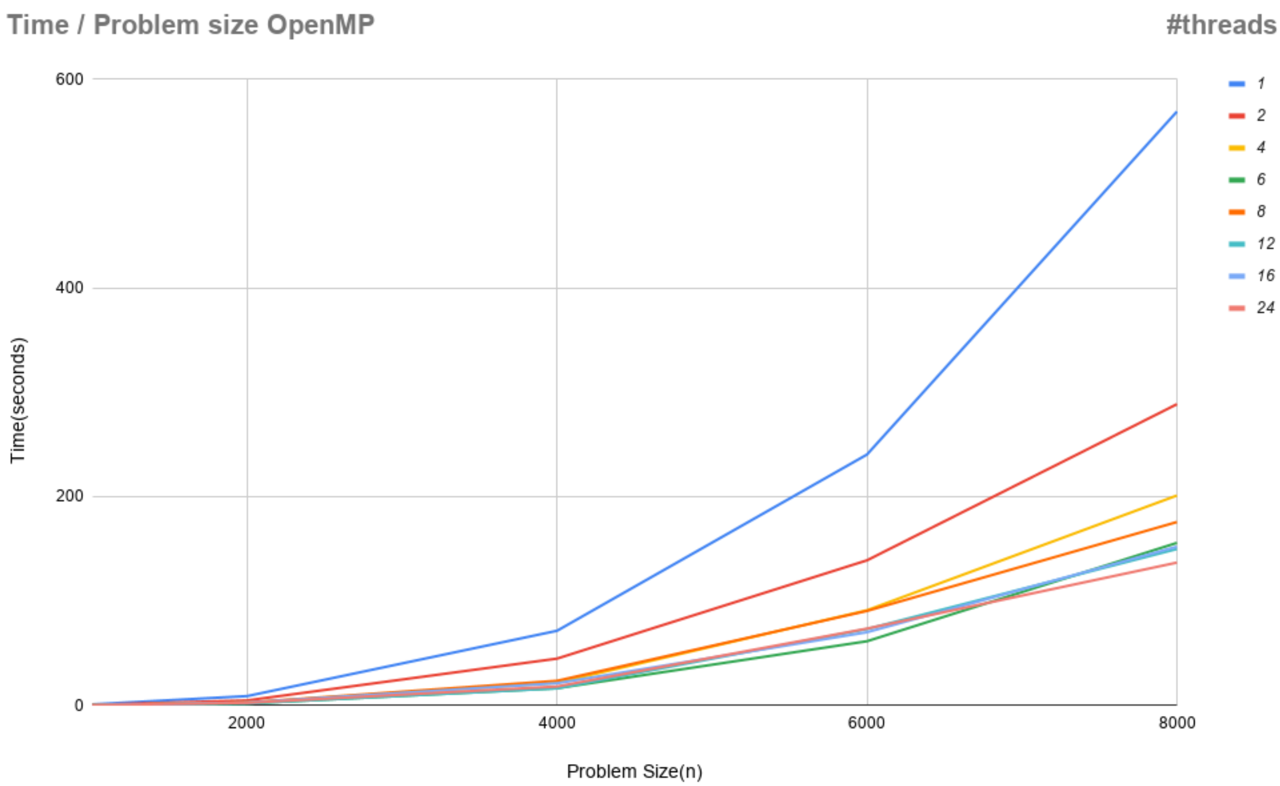

Efficiency vs Number of Processors Pthreads (n = 8000)
0.8
0.6
0.4
0.2
0
Efficiency
5
10
15
20
Number of Processors


Efficiency vs Number of Processors OpenMP (n=8000)
1
0.75
0.5
0.25
0
Efficiency
5
10
15
20
Number of Processors


Speedup vs. Number of Processors Pthread (n=8000)
Speedup
Number of Processors
0
2
4
6
5
10
15
20


Speedup vs. Number of Processors OpenMP (n=8000)
Speedup
Number of Processors
0
1
2
3
4
5
5
10
15
20


Time / Problem size Pthreads
#threads
800
600
400
200
0
2000
4000
6000
8000
Time(seconds)
Problem Size(n)
1
2
4
6
8
12
16
24



Time / Problem size OpenMP
#threads
1
2
4
6
8
12
16
24
Time(seconds)
600
400
200
0
2000
4000
6000
8000
Problem Size(n)