PARALLEL AND DISTRIBUTED COMPUTING
L – 19,20
DATE: 12.9.19
FACULTY : PROF. GAYATHRI R.
DHRUBANKA DUTTA, 17BCE1019

Implement a sudoku solver using mpi.

CODE :

```
#include "sudoku.h"
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int empty_cells=0;

struct pair_t{
	int x,y;
};
typedef struct pair_t pair;
pair empty_cells_list[SIZE*SIZE];

struct queue_element{
	int board[SIZE][SIZE];
	int forward_map[SIZE][SIZE][SIZE];
};

struct queue
{
	struct queue_element **list;
	int size,length;
	int head,tail;
};

int forward_map[SIZE][SIZE][SIZE];
int found = 0;
int **final_board;
int reverse_map_row[SIZE][SIZE][SIZE],reverse_map_column[SIZE][SIZE][SIZE];
int reverse_map_box[SIZE][SIZE][MINIGRIDSIZE][MINIGRIDSIZE];

void init_queue(struct queue *q, int length)
{
	q->list = malloc(sizeof(struct queue_element*)*length);
	q->head=0;
	q->tail=0;
	q->length=0;
```

```
	q->size=length;
}

void push(struct queue *q, struct queue_element *elem)
{
	if((q->tail+1)%q->size==(q->head))
	{
		printf("Queue full!!! PAAANIIIC\n");
		exit(0);
	}
	q->list[q->tail] = elem;
	q->tail = ((q->tail)+1)%(q->size);
	q->length++;
}

struct queue_element* pop(struct queue *q)
{
	if(q->head == q->tail)
	{
		printf("Queue empty!!! PAAANIIIC\n");
		exit(0);
	}
	struct queue_element* ret = q->list[q->head];
	q->head = ((q->head)+1)%(q->size);
	q->length--;
}

void populate_f(int x, int y, int val, int map[SIZE][SIZE][SIZE])
{
	if(val!=0)
	{
		int j,k;
		for(j=0;j<SIZE;j++)
		{
			if(j!=x)
				map[j][y][val-1]=1;
			if(j!=y)
				map[x][j][val-1]=1;
			if(j!=val-1)
				map[x][y][j]=1;
		}
		j=x-x%MINIGRIDSIZE;
		k=y-y%MINIGRIDSIZE;

		int jj,kk;
		for(jj=j;jj<j+MINIGRIDSIZE;jj++)
			for(kk=k;kk<k+MINIGRIDSIZE;kk++)
				if(jj!=x || kk!=y)
					map[jj][kk][val-1]=1;
```

```
	}
}

void dfs_populate_f(int x, int y, int val, int map[SIZE][SIZE][SIZE])
{
	if(val!=0)
	{
		int j,k;
		for(j=0;j<SIZE;j++)
		{
			if(j!=x)
				map[j][y][val-1]=1;
			if(j!=y)
				map[x][j][val-1]=1;
			// if(j!=val-1)
			//	map[x][y][j]=1;
		}
		j=x-x%MINIGRIDSIZE;
		k=y-y%MINIGRIDSIZE;

		int jj,kk;
		for(jj=j;jj<j+MINIGRIDSIZE;jj++)
			for(kk=k;kk<k+MINIGRIDSIZE;kk++)
				if(jj!=x || kk!=y)
					map[jj][kk][val-1]=1;

	}
}

int findPosition(int x, int y)
{
	int i,single=0,idx=-1;
	for(i=0; i<SIZE; i++)
	{
		if(single)
		{
			if(forward_map[x][y][i] == 0)
				return -1;
		}
		else
		{
			if(forward_map[x][y][i] == 0)
			{
				single = 1;
				idx = i+1;
			}
		}
	}
```

```
	return idx;
}

int findCol(int x, int y, int **inp)
{
	int i,single=0,idx=-1;
	for(i=0; i<SIZE; i++)
	{
		if(single)
		{
			if(forward_map[x][i][y] == 0 && inp[x][i]==0)
				return -1;
		}
		else
		{
			if(forward_map[x][i][y] == 0 && inp[x][i]==0)
			{
				single = 1;
				idx = i;
			}
		}
	}
	return idx;
}

int findRow(int x, int y, int **inp)
{
	int i,single=0,idx=-1;
	for(i=0; i<SIZE; i++)
	{
		if(single)
		{
			if(forward_map[i][x][y] == 0 && inp[i][x]==0)
				return -1;
		}
		else
		{
			if(forward_map[i][x][y] == 0 && inp[i][x]==0)
			{
				single = 1;
				idx = i;
			}
		}
	}
	return idx;
}

pair findCell(int x, int y, int val, int **inp)
{
```

```
	int i,j,single=0;
	pair idx;
	idx.x=-1;
	for(i=x; i<x+MINIGRIDSIZE; i++)
	{
		for(j=y; j<y+MINIGRIDSIZE; j++)
		{
			if(single)
			{
				if(forward_map[i][j][val] == 0 && inp[i][j]==0)
				{
					idx.x=-1;
					idx.y=-1;
					return idx;
				}
			}
			else
			{
				if(forward_map[i][j][val] == 0 && inp[i][j]==0)
				{
					single = 1;
					idx.x = i;
					idx.y = j;
				}
			}
		}
	}
	return idx;
}

void dfs(int board[SIZE][SIZE], int forward_map[SIZE][SIZE][SIZE], int idx)
{
	// printf("idx=%d\n", idx);
	if(idx==empty_cells)
	{
		printf("found!!\n");
		found = 1;
		int i,j;
		for(i=0;i<SIZE;i++)
		{
			for(j=0;j<SIZE;j++)
				printf("%d ",board[i][j]);
			printf("\n");
		}
		// printf("sizeof(board)=%lx\n",sizeof(board));
		// memcpy(final_board,board,sizeof(board));
		for(i=0;i<SIZE;i++)
			for(j=0;j<SIZE;j++)
				final_board[i][j]=board[i][j];
```

```
            // printf("returning\n");
            return;
        }
        else if(found)
            return;
        else
        {
            int val;
            for(val=0; val<SIZE; val++)
            {
                int x = empty_cells_list[idx].x;
                int y = empty_cells_list[idx].y;
                if(forward_map[x][y][val]==0)
                {
                    int row_bkp[SIZE], col_bkp[SIZE], box_bkp[MINIGRIDSIZE]
[MINIGRIDSIZE], val_bkp[SIZE];

                    int j,k;
                    for(j=0;j<SIZE;j++)
                    {
                        if(j!=x)
                            col_bkp[j]=forward_map[j][y][val];
                        if(j!=y)
                            row_bkp[j]=forward_map[x][j][val];
                    }
                    j=x-x%MINIGRIDSIZE;
                    k=y-y%MINIGRIDSIZE;

                    int jj,kk;
                    for(jj=j;jj<j+MINIGRIDSIZE;jj++)
                        for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                            if(jj!=x || kk!=y)
                                box_bkp[jj-j][kk-k]=forward_map[jj][kk][val];

                    board[x][y]=val+1;
                    dfs_populate_f(x,y,val+1,forward_map);
                    dfs(board,forward_map,idx+1);

                    for(j=0;j<SIZE;j++)
                    {
                        if(j!=x)
                            forward_map[j][y][val]=col_bkp[j];
                        if(j!=y)
                            forward_map[x][j][val]=row_bkp[j];
                    }
                    j=x-x%MINIGRIDSIZE;
                    k=y-y%MINIGRIDSIZE;

                    for(jj=j;jj<j+MINIGRIDSIZE;jj++)
```

```
                                        for(kk=k;kk<k+MINIGRIDSIZE;kk++)
                                                if(jj!=x || kk!=y)
                                                        forward_map[jj][kk][val]=box_bkp[jj-j][kk-k];

                                board[x][y]=0;
                        }
                }
        }
}

int **solveSudoku(int ** inp)
{
        final_board=malloc(sizeof(int*)*SIZE);
        int i,j;
        for(i=0;i<SIZE;i++)
                final_board[i]=malloc(sizeof(int)*SIZE);
        memset(forward_map,0,sizeof(forward_map));
        memset(reverse_map_row,0,sizeof(reverse_map_row));
        memset(reverse_map_column,0,sizeof(reverse_map_column));
        memset(reverse_map_box,0,sizeof(reverse_map_box));


        //#pragma omp parallel for
        for(i=0;i<SIZE*SIZE;i++)
        {
                // printf("i=%d\n", i);
                int x=i%SIZE;
                int y=i/SIZE;
                int val=inp[x][y];
                if(val>0)
                        populate_f(x,y,val,forward_map);
        }
        int changed_outer=1;
        while(changed_outer)
        {
                int changed=1;
                changed_outer=0;
                while(changed)
                {
                        changed=0;
                        // #pragma omp parallel
                        {
        //ELIMINATION
                                // #pragma omp for
                                for(j=0;j<SIZE*SIZE;j++)
                                {
                                        int x=j%SIZE, y=j/SIZE;
                                        if(inp[x][y]!=0)
                                        {
```

```
                                        // printf("x=%d,y=%d\n", x,y);
                                        continue;
                                }
                                int pos = findPosition(x,y);
                                if(pos>0)
                                {
                                        inp[x][y]=pos;
                                        printf("Elimination at x=%d, y=%d, val=%d\n", x,y,pos);
                                        populate_f(x,y,pos,forward_map);
                                        changed=1;
                                }
                        }
                        // #pragma omp for
                        for(j=0;j<SIZE;j++)
                        {
                                int p;
                                for(p=0;p<SIZE;p++)
                                {
        //LONE-RANGER-ROW
                                        int pos = findCol(j,p,inp);
                                        if(pos>=0)
                                        {
                                                inp[j][pos]=p+1;
                                                printf("Lone ranger 1 at x=%d, y=%d, val=%d\n",
j,pos,p+1);
                                                populate_f(j,pos,p+1,forward_map);
                                                changed=1;
                                        }

        //LONE-RANGER-COL
                                        pos = findRow(j,p,inp);
                                        if(pos>=0)
                                        {
                                                inp[pos][j]=p+1;
                                                printf("Lone ranger 2 at x=%d, y=%d, val=%d\n",
pos,j,p+1);
                                                populate_f(pos,j,p+1,forward_map);
                                                changed=1;
                                        }

        //LONE_RANGER-BOX
                                        int x=(j%MINIGRIDSIZE)*MINIGRIDSIZE,
y=(j/MINIGRIDSIZE)*MINIGRIDSIZE;
                                        pair p1 = findCell(x,y,p,inp);
                                        if(p1.x>=0)
                                        {
                                                inp[p1.x][p1.y]=p+1;
                                                printf("Lone ranger 3 at x=%d, y=%d, val=%d\n",
p1.x,p1.y,p+1);
```

```
                                        populate_f(p1.x,p1.y,p+1,forward_map);
                                        changed=1;
                                }
                        }
                }

        }
    }
// TWINS ROWS
    int k;
    for(k=0;k<SIZE;k++)
    {
        for(i=0;i<SIZE;i++)
        {
            for(j=i+1;j<SIZE;j++)
            {
                int l,cnt=0,first=0,second=0,position[2];
                for(l=0;l<SIZE;l++)
                {
                    if(forward_map[k][l][i]==0){
                        first|=1<<l;
                        if(cnt<=1)
                            position[cnt]=l;
                        else
                        {
                            cnt=0;
                            break;
                        }
                        cnt++;

                    }
                    if(forward_map[k][l][j]==0)
                        second|=1<<l;
                }
                if(cnt==2 && second==first)
                {
                    for(l=0;l<SIZE;l++)
                    {
                        forward_map[k][position[0]][l]=1;
                        forward_map[k][position[1]][l]=1;
                    }
                    forward_map[k][position[0]][j]=0;
                    forward_map[k][position[0]][i]=0;
                    forward_map[k][position[1]][j]=0;
                    forward_map[k][position[1]][i]=0;
                }
            }
        }
    }
```

```
// TWINS COLUMNS
        for(k=0;k<SIZE;k++)
        {
                for(i=0;i<SIZE;i++)
                {
                        for(j=i+1;j<SIZE;j++)
                        {
                                int l,cnt=0,first=0,second=0,position[2];
                                for(l=0;l<SIZE;l++)
                                {
                                        if(forward_map[l][k][i]==0){
                                                first|=1<<l;
                                                if(cnt<=1)
                                                        position[cnt]=l;
                                                else
                                                {
                                                        cnt=0;
                                                        break;
                                                }
                                                cnt++;

                                        }
                                        if(forward_map[l][k][j]==0)
                                                second|=1<<l;
                                }
                                if(cnt==2 && second==first)
                                {
                                        for(l=0;l<SIZE;l++)
                                        {
                                                forward_map[position[0]][k][l]=1;
                                                forward_map[position[1]][k][l]=1;
                                        }
                                        forward_map[position[0]][k][j]=0;
                                        forward_map[position[0]][k][i]=0;
                                        forward_map[position[1]][k][j]=0;
                                        forward_map[position[1]][k][i]=0;
                                }
                        }
                }
        }
        // TWINS GRID

        for(k=0;k<SIZE;k++)
        {
                int x=(k%MINIGRIDSIZE)*MINIGRIDSIZE,
y=(k/MINIGRIDSIZE)*MINIGRIDSIZE;
                for(i=0;i<SIZE;i++)
                {
                        for(j=i+1;j<SIZE;j++)
```

```
                        {
                                int l1,l2,cnt=0,first=0,second=0,positionx[2],positiony[2];
                                for(l1=x;l1<x+MINIGRIDSIZE;l1++)
                                        for(l2=y;l2<y+MINIGRIDSIZE;l2++)
                                        {
                                                if(forward_map[l1][l2][i]==0){
                                                        first|=1<<(l1*MINIGRIDSIZE+l2);
                                                        if(cnt<=1)
                                                        {
                                                                positionx[cnt]=l1;
                                                                positiony[cnt]=l2;
                                                        }
                                                        else
                                                        {
                                                                cnt=0;
                                                                break;
                                                        }
                                                        cnt++;

                                                }
                                                if(forward_map[l1][l2][j]==0)
                                                        second|=1<<(l1*MINIGRIDSIZE+l2);
                                        }
                                if(cnt==2 && second==first)
                                {
                                        for(l1=0;l1<SIZE;l1++)
                                        {
                                                forward_map[positionx[0]][positiony[0]][l1]=1;
                                                forward_map[positionx[1]][positiony[1]][l1]=1;
                                        }
                                        forward_map[positionx[0]][positiony[0]][j]=0;
                                        forward_map[positionx[0]][positiony[0]][i]=0;
                                        forward_map[positionx[1]][positiony[1]][j]=0;
                                        forward_map[positionx[1]][positiony[1]][i]=0;
                                }
                        }
                }
        }
}
final_board=inp;
for(i=0;i<SIZE;i++)
{
        for(j=0;j<SIZE;j++)
                printf("%d ",inp[i][j]);
        printf("\n");
}

// #pragma omp parallel for
for(i=0; i<SIZE*SIZE; i++)
```

```
{
	int x=i%SIZE, y=i/SIZE;
	if(inp[x][y]==0)
	{
		int ind;
			// #pragma omp critical
			ind=empty_cells++;
		empty_cells_list[ind].x=x;
		empty_cells_list[ind].y=y;
	}
}
printf("empty cells %d\n",empty_cells);
struct queue *q = malloc(sizeof(struct queue));
int num_threads;
int idx=0;
#pragma omp parallel shared(q)
{

	int k;
	#pragma omp single
	{
		num_threads = omp_get_num_threads();
		init_queue(q, num_threads*SIZE);
		struct queue_element *elem;
		elem = malloc(sizeof(struct queue_element));
		// memcpy((elem->board),(inp),sizeof(inp));
		for(i=0;i<SIZE;i++)
			for(j=0;j<SIZE;j++)
				elem->board[i][j]=inp[i][j];

		// memcpy(elem->forward_map,forward_map,sizeof(forward_map));
		for(i=0;i<SIZE;i++)
			for(j=0;j<SIZE;j++)
				for(k=0;k<SIZE;k++)
					elem->forward_map[i][j][k]=forward_map[i][j][k];

		push(q,elem);
		while(q->length < num_threads && idx<empty_cells)
		{
			int l=q->length;
			int i;
			struct queue *tmp=malloc(sizeof(struct queue));
			init_queue(tmp,l*SIZE);
			for(i=0;i<l;i++)
			{
				int j;
				for(j=0;j<SIZE;j++)
				{
```

```
                                        if(forward_map[empty_cells_list[idx].x]
[empty_cells_list[idx].y][j]==0)
                                        {
                                                int a,b,c;
                                                struct queue_element * temp=malloc(sizeof(struct
queue_element));
                                                // memcpy(temp->board,q->list[i]-
>board,sizeof(temp->board));
                                                for(a=0;a<SIZE;a++)
                                                        for(b=0;b<SIZE;b++)
                                                                temp->board[a][b]=q->list[i]-
>board[a][b];


                                                // memcpy(temp->forward_map,q->list[i]-
>forward_map,sizeof(temp->forward_map));
                                                for(a=0;a<SIZE;a++)
                                                        for(b=0;b<SIZE;b++)
                                                                for(c=0;c<SIZE;c++)
                                                                        temp->forward_map[a][b]
[c]=q->list[i]->forward_map[a][b][c];

                                                temp->board[empty_cells_list[idx].x]
[empty_cells_list[idx].y]=j+1;

        populate_f(empty_cells_list[idx].x,empty_cells_list[idx].y,j+1,temp->forward_map);
                                                push(tmp,temp);
                                        }
                                }
                        }
                        idx++;
                        free(q);
                        q=tmp;
                }

        }
        // init_queue(&q, );
        #pragma omp for schedule(dynamic,1)
        for(i=0; i<q->length;i++)
        {
                printf("dfs i=%d\n", i);
                // for(k=0;k<SIZE;k++)
                // {
                //      for(j=0;j<SIZE;j++)
                //              printf("%d ",q->list[i]->board[k][j]);
                //      printf("\n");
                // }

                dfs(q->list[i]->board,q->list[i]->forward_map,idx);
```

```
                }
        }
//BRUTE-FORCE
        for(i=0;i<SIZE;i++)
        {
                for(j=0;j<SIZE;j++)
                        printf("%d ",final_board[i][j]);
                printf("\n");
        }
        return final_board;
}
```

OUTPUT :