Dependency Analysis

Dependencies Affect Parallelization!

- Data must be produced and consumed in the *correct order*

- Simple example of data dependence:

```
S1  PI = 3.14
S2  R = 5.0
S3  AREA = PI * R ** 2
```

- Statement *S3* cannot be moved before either *S1* or *S2* without compromising correct results

Dependencies Affect Parallelization!

- Dependency analysis identifies the units of code that are semantically independent and hence executable in parallel.

- Modern compilers do provide some degree of optimization.

Categorizing Dependencies

- There are 3 categories of program dependencies:
  - Control Flow Dependencies
  - Data Flow Dependencies
  - Loop Dependencies

Categorizing Dependencies

- ***Control Dependency:*** How different program instructions affect each other.

- ***Data Dependency:*** How different pieces of data are related and affect each other during execution.

Control Dependencies

- Every program has a well-defined flow of control.
- This flow can be affected by several kinds of operations such as Loops and branches

- Control flow dependencies are those statements that are only executed if the program flow-of-control allows its execution.

Control Flow Dependency

Example:

```
S1      if (x > 100) {
S2          x = 100;
        }
S3      y = x;
```

- The value of **y** depends on what the condition **(x > 100)** evaluates to.

S2 is control-flow dependent on S1 written:

$$S1\ \boldsymbol{\delta}^{\mathbf{c}}\ S2$$

Data Flow Dependency

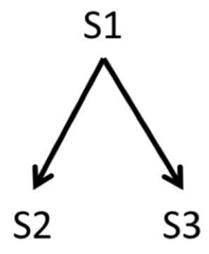
- Data flow dependencies arise from reading or writing to the same variable or resource.
- There are 4-types of data dependencies:
  - Flow, Anti, Output and Input

Flow (True) Dependence

- A statement S_j is flow dependent on S_i if and only if S_i modifies a resource that S_j requires and S_i precedes it in execution.

Example:

S1	A = B + C;
S2	D = A + 2;
S3	E = A * 3;



S2 and S3 are therefore flow dependent on S1 written:

S2 δ_f S1

S3 δ_f S1

Anti-Dependence

- A statement S_j is anti-dependent on S_i if and only if S_j modifies a resource that S_i requires and S_i precedes it in execution.

Example:	
S1	A = B + C;
S2	B = D / 2;

S2 is therefore anti-dependent on S1

$\boldsymbol{S1\ \delta^{a}\ S2}$

Output-Dependence

- A statement S_j is output dependent on S_i if and only if S_i and S_j modify the same resource and S_i precedes S_j in execution.

Example:

```
S1        X = B + C;
S2        X = D / 2;
```

S2 is therefore output-dependent on S1

S1 δ^O S2

Input-Dependence

- A statement S_j is input dependent on S_i if and only if S_i and S_j read the same resource and S_i precedes S_j in execution.
- This type of dependence does not prohibit reordering of statements

Example:

```
S1        X = B + C;
S2        Y = B / 2;
```

S2 is therefore input-dependent on S1

$\boldsymbol{S1\ \delta^{i}\ S2}$

An example

```
int main(int argc, char** argv) {
    int x,y;
    int z=10;
    x=TaskX();                    //s1
    y=TaskY()+z;                  //s2
    y=x-5;                        //s3
    z=TaskZ();                    //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```

```
int TaskX() {
  sleep(15);
  return 15;}
int TaskY() {
  sleep(5);
  return 5; }
int TaskZ() {
  sleep(5);
  return 5; }
```

What would be the output?
How long would it take to produce output?

The output

```
ez@master: ~
$ gcc serial.c -o serial
$ time ./serial
x: 15 y: 10 z: 5

real    0m25.004s
user    0m0.000s
sys     0m0.000s
$ 
```

How it can be improved? We know how to implement the parallel version using omp or mpi, however …

The parallel version

- Let us assume we intend to use four processors for improved performance
- Ideally, each statement can run independently and we have speedup and efficiency of 4 and 1 respectively.
- Here is a possible (not optimized) implementation.
  - The processes 1-4, execute the individual statements,S1-S4 and send the results to process 0
  - The process 0 receives the values, updates the variables and displays the results
  - The objective is to mimic the output of serial code, just a rough implementation

The parallel version

```
switch(myrank){
   case 1:
    x=TaskX(); //s1
    MPI_Send(&x,1,MPI_INT,0,787,MPI_COMM_WORLD);
    break;
   case 2:
    y=TaskY()+z; //s2
    MPI_Send(&y,1,MPI_INT,0,787,MPI_COMM_WORLD);
    break;
   case 3:

            ...
  }
```

The parallel version

```
if (myrank == 0) {
    MPI_Status status;
    int recvdValue,i;
    for (i=1;i<5;i++) {
      MPI_Recv(&recvdValue, 1, MPI_INT,
            MPI_ANY_SOURCE, 787, MPI_COMM_WORLD, &status);
      switch(status.MPI_SOURCE){
        case 1:
          x=recvdValue;
          break;
        case 2:
          y=recvdValue;

              ...
```

The output

```
ez@master: ~                                          ×   ez@maste
$ gcc serial.c -o serial
$ time ./serial
x: 15 y: 10 z: 5

real    0m25.002s
user    0m0.000s
sys     0m0.000s
$
$
$
$ mpicc parallel.c -o parallel
$ time mpiexec -n 5 -f machinefile ./parallel
x: 15 y: 15 z: 5

real    0m15.225s
user    0m14.412s
sys     0m0.348s
$
```

The output

- Few interesting things to notice
  - The time values are overall just but not the breakdown is
  - The values of the variables are incorrect when compared to the serial code
  - What can be the reason?

Dependencies

```
int main(int argc, char** argv) {
   int x,y;
   int z=10;
   x=TaskX();                    //s1
   y=TaskY()+z;         //s2
   y=x-5;               //s3
   z=TaskZ();                    //s4

   printf("x: %d y: %d z: %d\n", x, y, z);
   return 0;
}
```

Is there any dependence relation between s3 and s1?
What can be done? Can it be somehow removed?

The parallel version – v2

```
switch(myrank){
  case 1:
    x=TaskX();
    MPI_Send(&x,1,MPI_INT,0,787,MPI_COMM_WORLD);
    y=x-5;
    MPI_Send(&y,1,MPI_INT,0,787,MPI_COMM_WORLD);
    break;
  case 2:
    y=TaskY()+z;
    MPI_Send(&y,1,MPI_INT,0,787,MPI_COMM_WORLD);
    break;
  case 3:
    z=TaskZ();
    MPI_Send(&z,1,MPI_INT,0,787,MPI_COMM_WORLD);
    break;
}
```

The parallel version – v2

```
if (myrank == 0) {
      MPI_Status status;
      int recvdValue,i;
      for (i=1;i<4;i++) {
        MPI_Recv(&recvdValue, 1, MPI_INT,
                  MPI_ANY_SOURCE, 787, MPI_COMM_WORLD, &status);
        switch(status.MPI_SOURCE){
          case 1:
            x=recvdValue;
            MPI_Recv(&recvdValue, 1, MPI_INT, 1, 787,
                      MPI_COMM_WORLD, &status);
            y=recvdValue;
            break;

                …
```

How about the output?

```
$ mpicc parallel.c -o parallel
$ time mpiexec -n 4 -f machinefile ./parallel
x: 15 y: 15 z: 5
```

- We have decreased the level of parallelism but unfortunately, we are still not there!

Dependencies

```
int main(int argc, char** argv) {
    int x,y;
    int z=10;
    x=TaskX();                    //s1
    y=TaskY()+z;        //s2
    y=x-5;              //s3
    z=TaskZ();                    //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```

Identify the dependence relationship between s2 and s3?

The parallel version – v2

```
switch(myrank){
  case 1:
   x=TaskX();
   MPI_Send(&x,1,MPI_INT,0,787,MPI_COMM_WORLD);
   y=x-5;
   MPI_Send(&y,1,MPI_INT,0,787,MPI_COMM_WORLD);
   break;

        ...
```

What is our mistake then?

The output dependence

- There is an output dependence between s2 and s3.
- There are two options to fix them:
  - Further reduce the level of parallelism
  - As the are regarded as false dependencies, try to remove them!
  - We can use a technique called variable renaming

False Dependencies – Variable renaming

```
int main(int argc, char** argv) {
    int x,y;
    int z=10;
    x=TaskX();              //s1
    y=TaskY()+z; //s2
    y=x-5;                              //s3
    z=TaskZ();              //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```

False Dependencies – Variable renaming

```
int main(int argc, char** argv) {
    int x,y;
    int z=10;
    x=TaskX();          //s1
    y=TaskY()+z; //s2
    y=x-5;                    //s3
    z=TaskZ();          //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```

False Dependencies – Variable renaming

```
int main(int argc, char** argv) {
    int x,y, y2;
    int z=10;
    x=TaskX();          //s1
    y2=TaskY()+z;       //s2
    y=x-5;                      //s3
    z=TaskZ();          //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```

False Dependencies – Variable renaming

```
int main(int argc, char** argv) {
   int x,y, y2;
   int z=10;
   x=TaskX();              //s1
   y2=TaskY()+z;           //s2
   y=x-5;                                //s3
   z=TaskZ();              //s4

   printf("x: %d y: %d z: %d\n", x, y, z);
   return 0;
}
```

False Dependencies – Variable renaming

```
int main(int argc, char** argv) {
    int x,y, y2;
    int z=10;
    x=TaskX();              //s1
    y2=TaskY()+z;           //s2
    y=x-5;                              //s3
    z=TaskZ();              //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```

False Dependencies – Variable renaming

```
int main(int argc, char** argv) {
    int x,y, y2;
    int z, z2=10;
    x=TaskX();              //s1
    y2=TaskY()+z2;          //s2
    y=x-5;                          //s3
    z=TaskZ();              //s4

    printf("x: %d y: %d z: %d\n", x, y, z);
    return 0;
}
```