

Multithreaded programming



CONCURRENCY PATTERNS

—
Dependencies and Futures



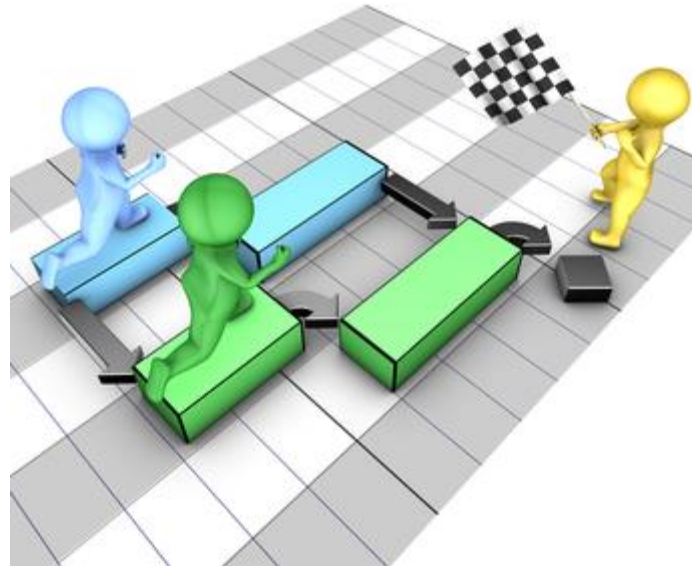
AARHUS
UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER
ENGINEERING

SWD
18 OCTOBER 2022

HENRIK BITSCH KIRK
ASSOCIATE PROFESSOR

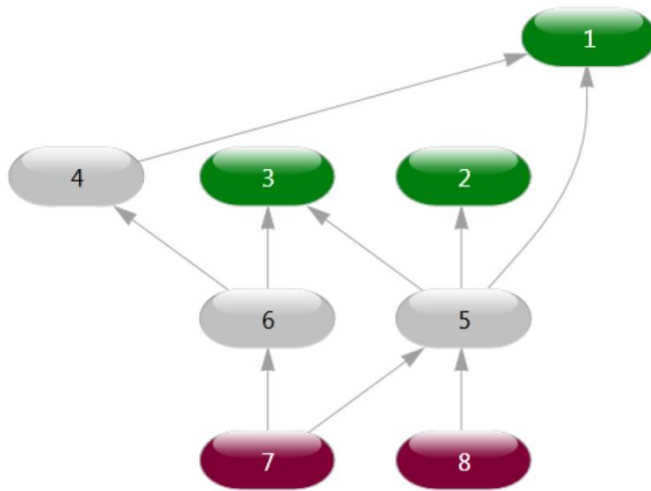


DEPENDENCIES



DEPENDENCIES

- Why are dependencies a problem to parallelism?

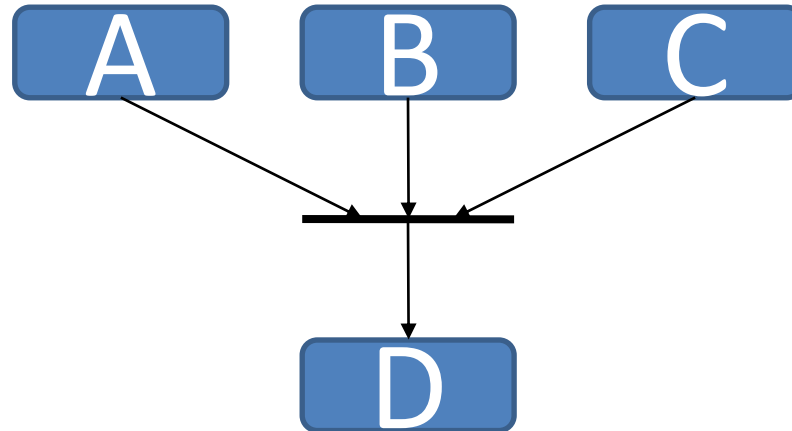


Directed Acyclic Graph (DAG)

- However, dependencies is a fact of life – so what should we do?

DEPENDENCIES IN DAG'S – ENTER CONTINUATIONS

- *Continuations* allow dependency synchronization to be handed over to TPL
- Specify a (set of) task(s) that must be completed before a new task starts:



```
var D = Task.Factory.ContinueWhenAll(A, B, C);
```

MAKE TIRAMISU

Recipe:

Begin by assembling four large egg yolks, 1/2 cup sweet marsala wine, 16 ounces mascarpone cheese, 12 ounces espresso, 2 tablespoons cocoa powder, 1 cup heavy cream, 1/2 cup granulated sugar, and enough lady fingers to layer a 12x8 inch pan twice (40).

Stir two tablespoons of granulated sugar into the espresso and put it in the refrigerator to chill.

Whisk the egg yolks

Pour in the sugar and wine and whisked briefly until it was well blended.

Pour some water into a saucepan and set it over high heat until it began to boil.

Lowering the heat to medium, place the heatproof bowl over the water and stirred as the mixture began to thicken and smooth out.

...

Link: <http://www.cookingforengineers.com/recipe/60/The-Classic-Tiramisu-original-recipe>

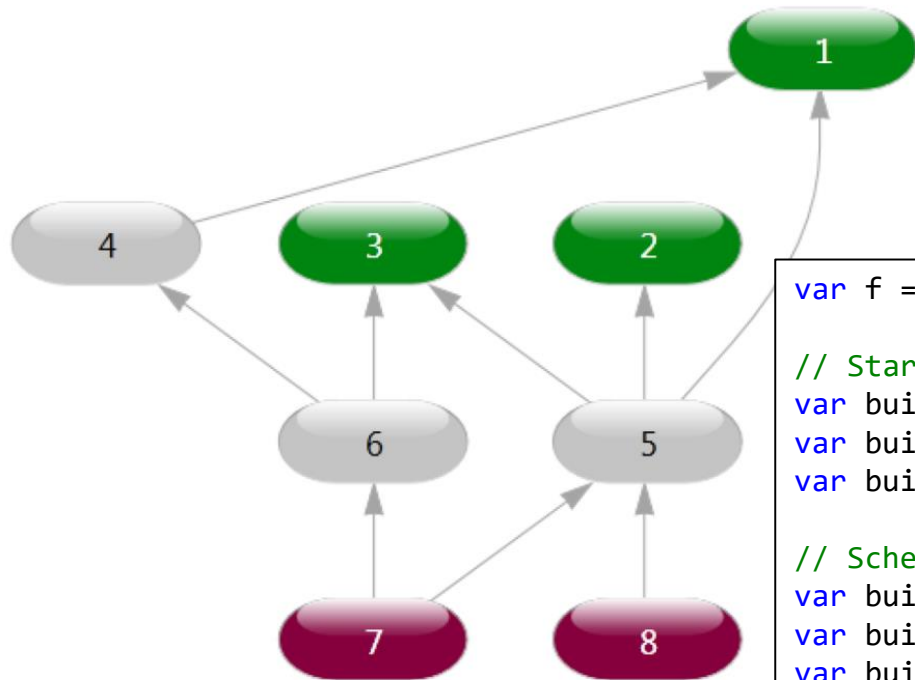


ALGORITHMYFIED

```
public Tiramisu make_tiramisu(eggs, sugar1, wine, cheese, cream, fingers, espresso, sugar2, cocoa)
{
    dissolve(sugar2, espresso);
    mixture = whisk(eggs);
    beat(mixture, sugar1, wine);
    whisk(mixture); // over steam
    whip(cream);
    beat(cheese);
    beat(mixture, cheese);
    fold(mixture, cream);
    assemble(mixture, fingers);
    sift(mixture, cocoa);
    refrigerate(mixture);
    return mixture; // it's now a tiramisu
}
```

What can be done in parallel?
Dependencies are hard to
find.

USING CONTINUATIONS TO WORK WITH DEPENDENCIES



```
var f = Task.Factory;

// Start 3 concurrent, independent builds
var build1 = f.StartNew(() => Build("project1"));
var build2 = f.StartNew(() => Build("project2"));
var build3 = f.StartNew(() => Build("project3"));

// Schedule continuations when dependent jobs are done
var build4 = f.ContinueWhenAll(new[] { build1 }, x => Build("project4"));
var build5 = f.ContinueWhenAll(new[] { build1, build2, build3 }, x => Build("project5"));
var build6 = f.ContinueWhenAll(new[] { build3, build4 }, x => Build("project6"));
var build7 = f.ContinueWhenAll(new[] { build5, build6 }, x => Build("project7"));
var build8 = f.ContinueWhenAll(new[] { build5 }, x => Build("project8"));

// Do work that is independent of builds
Console.WriteLine(System.DateTime.Now + "*** Doing build-independent work... ***");

Task.WaitAll(build1, build2, build3, build4, build5, build6, build7, build8);
```

```
var f = Task.Factory;
```

```
// Start 3 concurrent, independent builds
```

```
var build1 = f.StartNew(() => Build("project1"));
```

```
var build2 = f.StartNew(() => Build("project2"));
```

```
var build3 = f.StartNew(() => Build("project3"));
```

```
// Schedule continuations when dependent jobs are done
```

```
var build4 = f.ContinueWhenAll(new[] { build1 }, x => Build("project4"));
```

```
var build5 = f.ContinueWhenAll(new[] { build1, build2, build3 }, x => Build("project5"));
```

```
var build6 = f.ContinueWhenAll(new[] { build3, build4 }, x => Build("project6"));
```

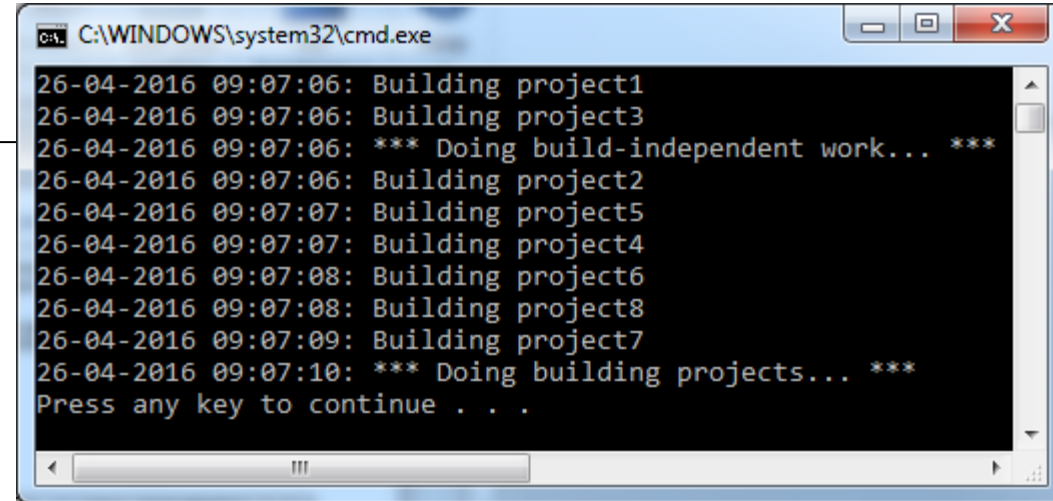
```
var build7 = f.ContinueWhenAll(new[] { build5, build6 }, x => Build("project7"));
```

```
var build8 = f.ContinueWhenAll(new[] { build5 }, x => Build("project8"));
```

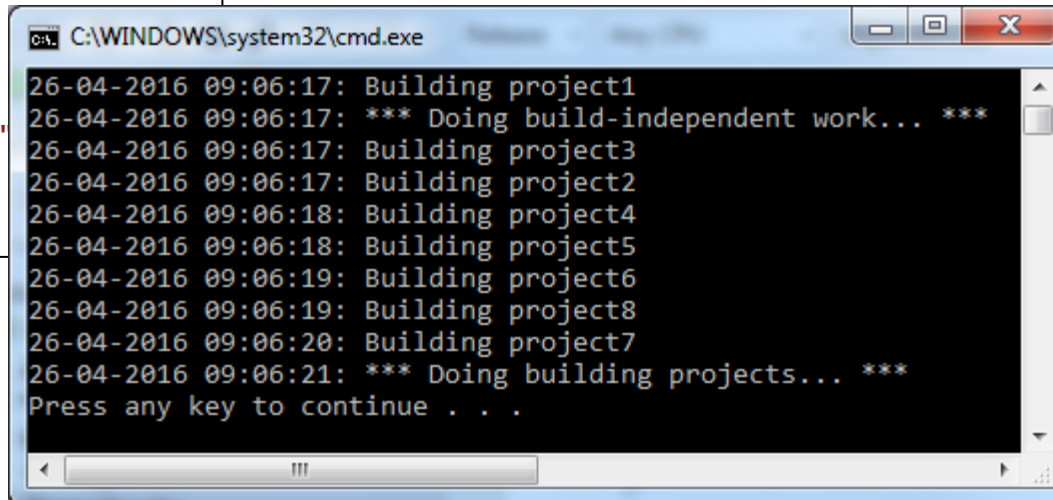
```
// Do work that is independent of builds
```

```
Console.WriteLine(System.DateTime.Now + "*** Doing build-independent work... ***")
```

```
Task.WaitAll(build1, build2, build3, build4, build5, build6, build7, build8);
```



```
C:\WINDOWS\system32\cmd.exe
26-04-2016 09:07:06: Building project1
26-04-2016 09:07:06: Building project3
26-04-2016 09:07:06: *** Doing build-independent work... ***
26-04-2016 09:07:06: Building project2
26-04-2016 09:07:07: Building project5
26-04-2016 09:07:07: Building project4
26-04-2016 09:07:08: Building project6
26-04-2016 09:07:08: Building project8
26-04-2016 09:07:09: Building project7
26-04-2016 09:07:10: *** Doing building projects... ***
Press any key to continue . . .
```



```
C:\WINDOWS\system32\cmd.exe
26-04-2016 09:06:17: Building project1
26-04-2016 09:06:17: *** Doing build-independent work... ***
26-04-2016 09:06:17: Building project3
26-04-2016 09:06:17: Building project2
26-04-2016 09:06:18: Building project4
26-04-2016 09:06:18: Building project5
26-04-2016 09:06:19: Building project6
26-04-2016 09:06:19: Building project8
26-04-2016 09:06:20: Building project7
26-04-2016 09:06:21: *** Doing building projects... ***
Press any key to continue . . .
```


FUTURES

—



AARHUS
UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER
ENGINEERING

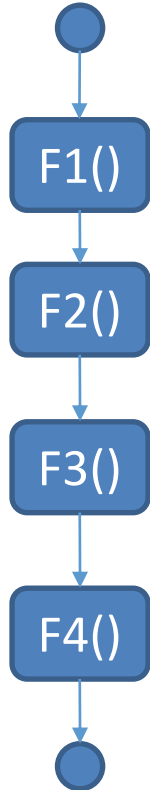
SWD



FUTURES – WHEN TASKS RETURN RESULTS

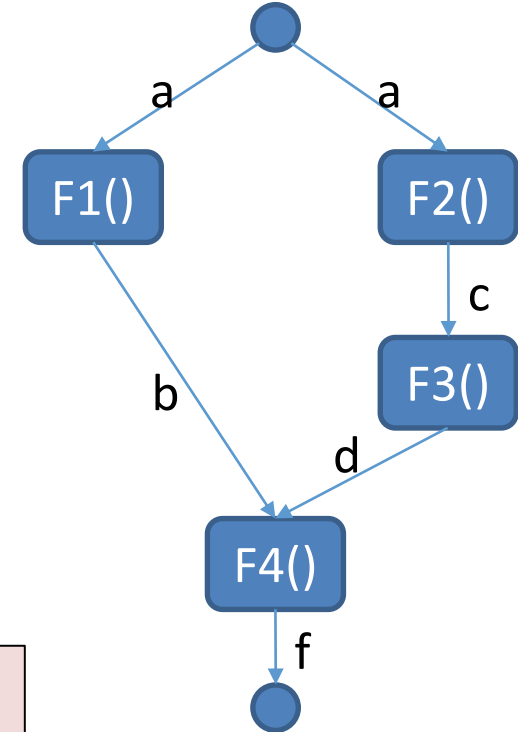
- A *future* is a stand-in for a computational result that is initially unknown but becomes available at a later time.
- The process of calculating the result can occur in parallel with other computations.
 - A future is a task that returns a *value* (`Task<TResult>`)
- Parallel tasks ~ *async actions* (no return value)
Futures ~ *async functions* (returning a value)
- Futures are used when we want to parallelize code with *data dependencies*.

AN EXAMPLE



```
static void Main(string[] args)
{
    var a = "A";
    var b = F1(a);
    var c = F2(a);
    var d = F3(c);
    var f = F4(b, d);
    System.Console.WriteLine(f);
}
```

Note how output of some functions are input to the next ones – this determines the potential parallelism!

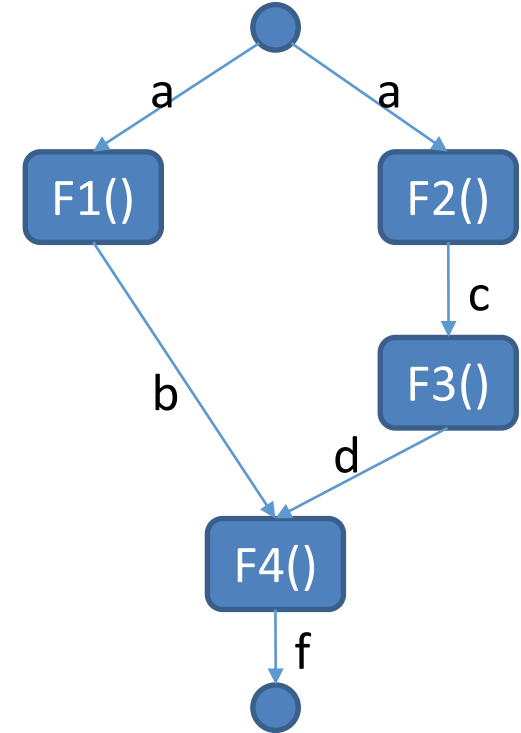


EXAMPLE CONT.

Parallelized using futures:

```
static void Main()
{
    var a = "A";
    Task<string> futureB = Task.Run(() => F1(a));
    var c = F2(a);
    var d = F3(c);
    var f = F4(futureB.Result, d);
    Console.WriteLine(f);
}
```

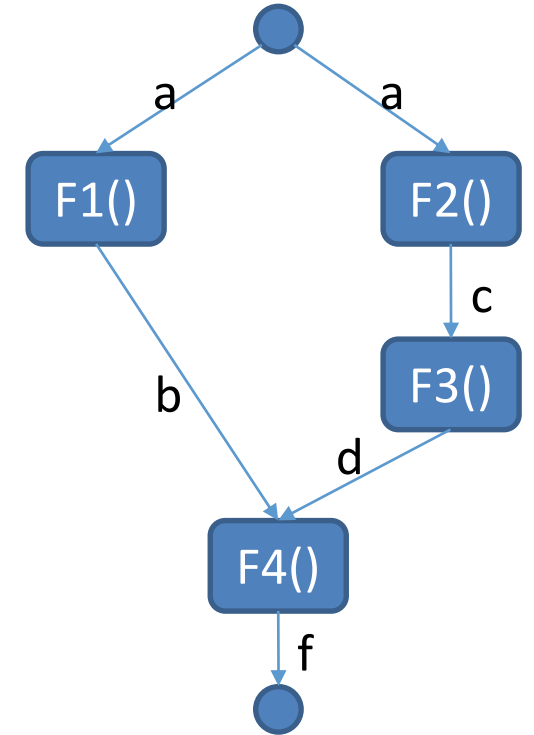
- The task runs the right part of the tree
- futureB is the "return value" from F1() and queried for the result as F4() is called



FUTURES - WHEN IS THE RESULT READY

- Futures are pretty darn clever!

```
static void Main()
{
    var a = "A";
    Task<string> futureB = Task.Run(() => F1(a));
    var c = F2(a);
    var d = F3(c);
    var f = F4(futureB.Result, d);
    Console.WriteLine(f);
}
```

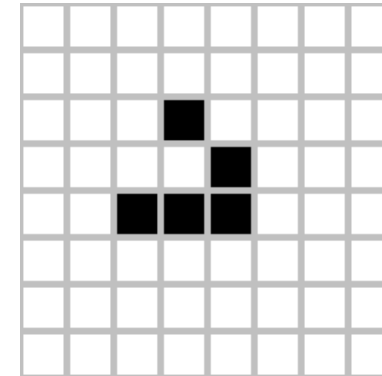


- By the time we query futureB for the result...
 - If task running F1() has *already* finished: futureB.Result is ready and immediately returned.
 - If task running F1() is *running but has not yet finished*: Calling thread blocks until futureB.Result is available.
 - If task running F1() *hasn't* started yet: The task will be executed inline in the current thread context, if possible.

DEPENDENCIES – ITERATING IN LOCK STEP

- A common algorithm pattern is to have iterations $[0..N)$, each consisting of many calculations, where iteration $i+1$ depends on calculations in iteration i .

- Examples:
 - Particle simulation
 - Heat dissipation
 - Conway's Game of Life



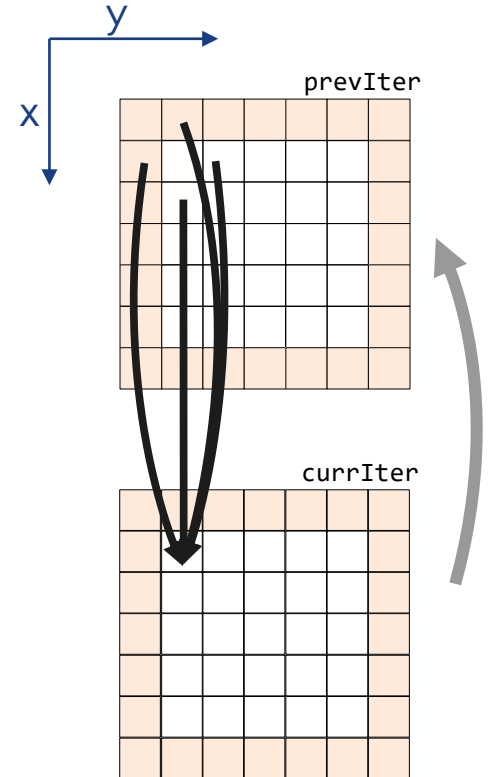
- In these cases, we cannot parallelize the iterations (as they are dependent), but we can parallelize the calculations *within* each calculation
- We must ensure *lock step*: All (parallel) calculations of one iteration have been completed before calculations in the next iteration start

HEAT DISSIPATION EXAMPLE - SEQUENTIAL

```
static double[,] SequentialSimulation(int plateSize, int timeSteps)
{
    // Initial plates for previous and current time steps, with
    // heat starting on one side
    var prevIter = new double[plateSize, plateSize];
    var currIter = new double[plateSize, plateSize];

    for (var y = 0; y < plateSize; y++) prevIter[y, 0] = 255.0f;

    // Run simulation
    for (int step = 0; step < timeSteps; step++)
    {
        for (int y = 1; y < plateSize - 1; y++)
        {
            for (int x = 1; x < plateSize - 1; x++)
            {
                currIter[y, x] =
                    ((prevIter[y, x - 1] +
                     prevIter[y, x + 1] +
                     prevIter[y - 1, x] +
                     prevIter[y + 1, x]) * 0.25f);
            }
        }
        Swap(ref prevIter, ref currIter);
    }
    return prevIter;
}
```

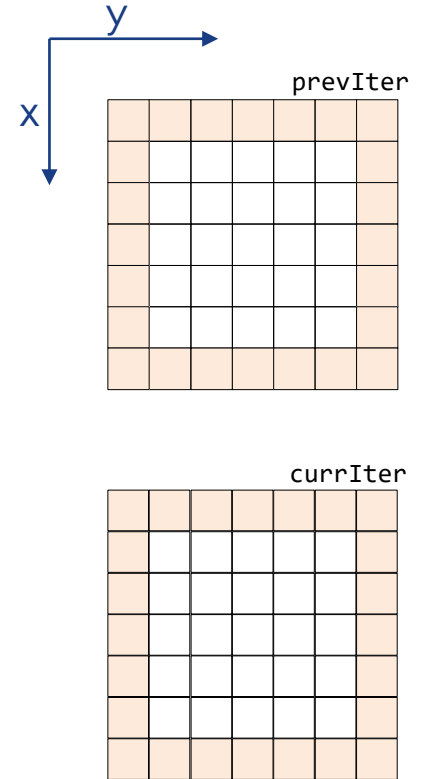


HEAT DISSIPATION - PARALLEL

```
static double[,] ParallelSimulation(int plateSize, int timeSteps)
{
    // Initial plates for previous and current time steps, with
    // heat starting on one side
    var prevIter = new double[plateSize, plateSize];
    var currIter = new double[plateSize, plateSize];

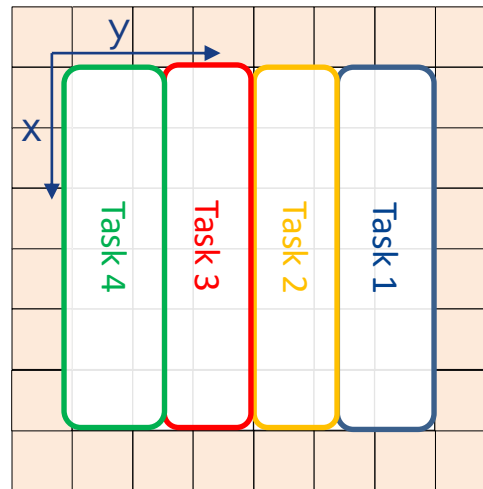
    for (var y = 0; y < plateSize; y++) prevIter[y, 0] = 255.0f;

    // Run simulation
    for (int step = 0; step < timeSteps; step++)
    {
        Parallel.For(1, plateSize - 1, y =>
        {
            for (int x = 1; x < plateSize - 1; x++)
            {
                currIter[y, x] =
                    ((prevIter[y, x - 1] +
                     prevIter[y, x + 1] +
                     prevIter[y - 1, x] +
                     prevIter[y + 1, x]) * 0.25f);
            }
        });
        Swap(ref prevIter, ref currIter);
    }
    return prevIter;
}
```



HEAT DISSIPATION – PARALLEL WITH TASK BARRIER

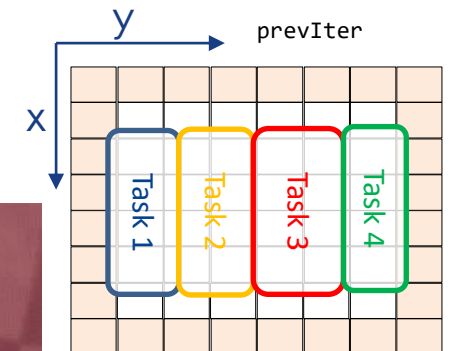
- To take advantage of cache locality, we can make sure that the same tasks always do calculations on the same section of the plate



HEAT DISSIPATION – PARALLEL WITH TASK BARRIER

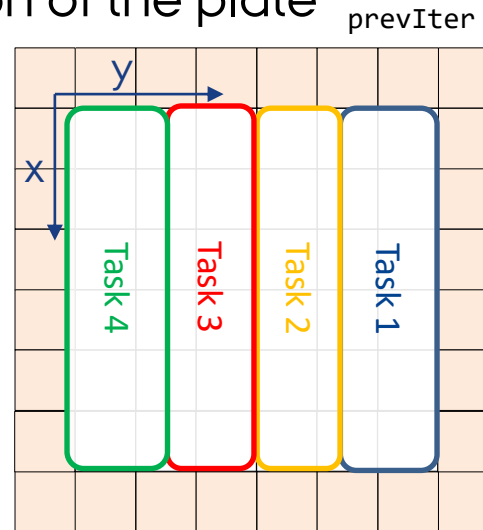
```
// Run simulation
int numTasks = Environment.ProcessorCount;
var tasks = new Task[numTasks];

int chunkSize = (plateSize - 2) / numTasks;
for (int i = 0; i < numTasks; i++)
{
    int yStart = 1 + (chunkSize * i);
    int yEnd = (i == numTasks - 1) ? plateSize - 1 : yStart + chunkSize;
    tasks[i] = Task.Run(() =>
    {
        for (int step = 0; step < timeSteps; step++)
        {
            for (int y = yStart; y < yEnd; y++)
            {
                for (int x = 1; x < plateSize - 1; x++)
                {
                    currIter[y, x] =
                        ((prevIter[y, x - 1] +
                          prevIter[y, x + 1] +
                          prevIter[y - 1, x] +
                          prevIter[y + 1, x]) * 0.25f);
                }
            }
        }
    });
}
```



HEAT DISSIPATION – PARALLEL WITH TASK BARRIER

- To take advantage of cache locality, we can make sure that the same tasks always do calculations on the same section of the plate



- We can then use a `System.Threading.Barrier` to synchronizing the tasks.

```
public Barrier(int signalCount) or  
public Barrier(int signalCount, Action<Barrier> postPhaseAction)  
  
Barrier.SignalAndWait()
```

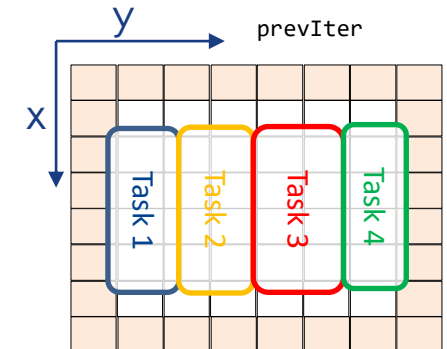
HEAT DISSIPATION – PARALLEL WITH TASK BARRIER

```
// Run simulation
int numTasks = Environment.ProcessorCount;
var tasks = new Task[numTasks];

var stepBarrier = new Barrier(numTasks, _ => Swap(ref prevIter, ref currIter));
int chunkSize = (plateSize - 2) / numTasks;
for (int i = 0; i < numTasks; i++)
{
    int yStart = 1 + (chunkSize * i);
    int yEnd = (i == numTasks - 1) ? plateSize - 1 : yStart + chunkSize;
    tasks[i] = Task.Run(() =>
    {
        for (int step = 0; step < timeSteps; step++)
        {
            for (int y = yStart; y < yEnd; y++)
            {
                for (int x = 1; x < plateSize - 1; x++)
                {
                    currIter[y, x] =
                        ((prevIter[y, x - 1] +
                          prevIter[y, x + 1] +
                          prevIter[y - 1, x] +
                          prevIter[y + 1, x]) * 0.25f);
                }
            }
            stepBarrier.SignalAndWait();
        }
    });
}
```

When numTasks tasks have reached barrier, do this action

The barrier



DEMO TIME

```
$ ./run  
Sequential: 10498  
Parallel: 19763  
Parallel with a barrier: 6033
```



AARHUS
UNIVERSITY