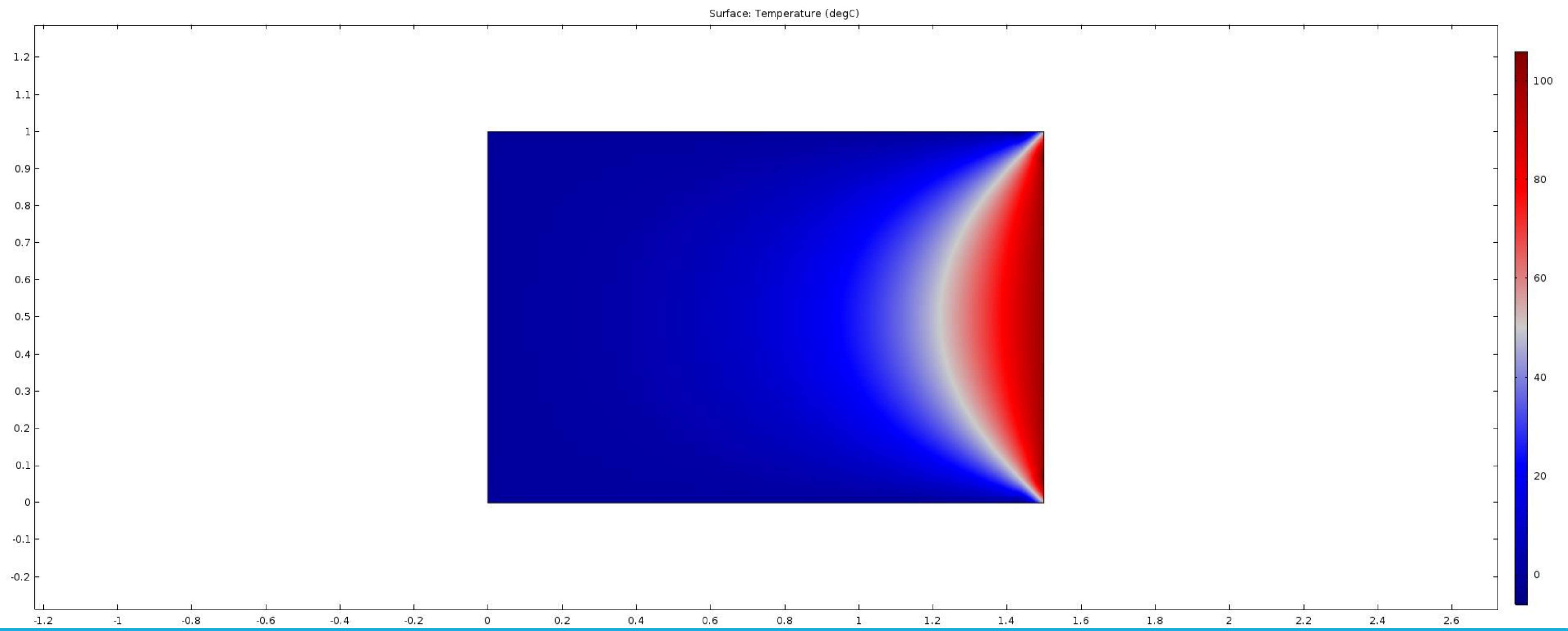
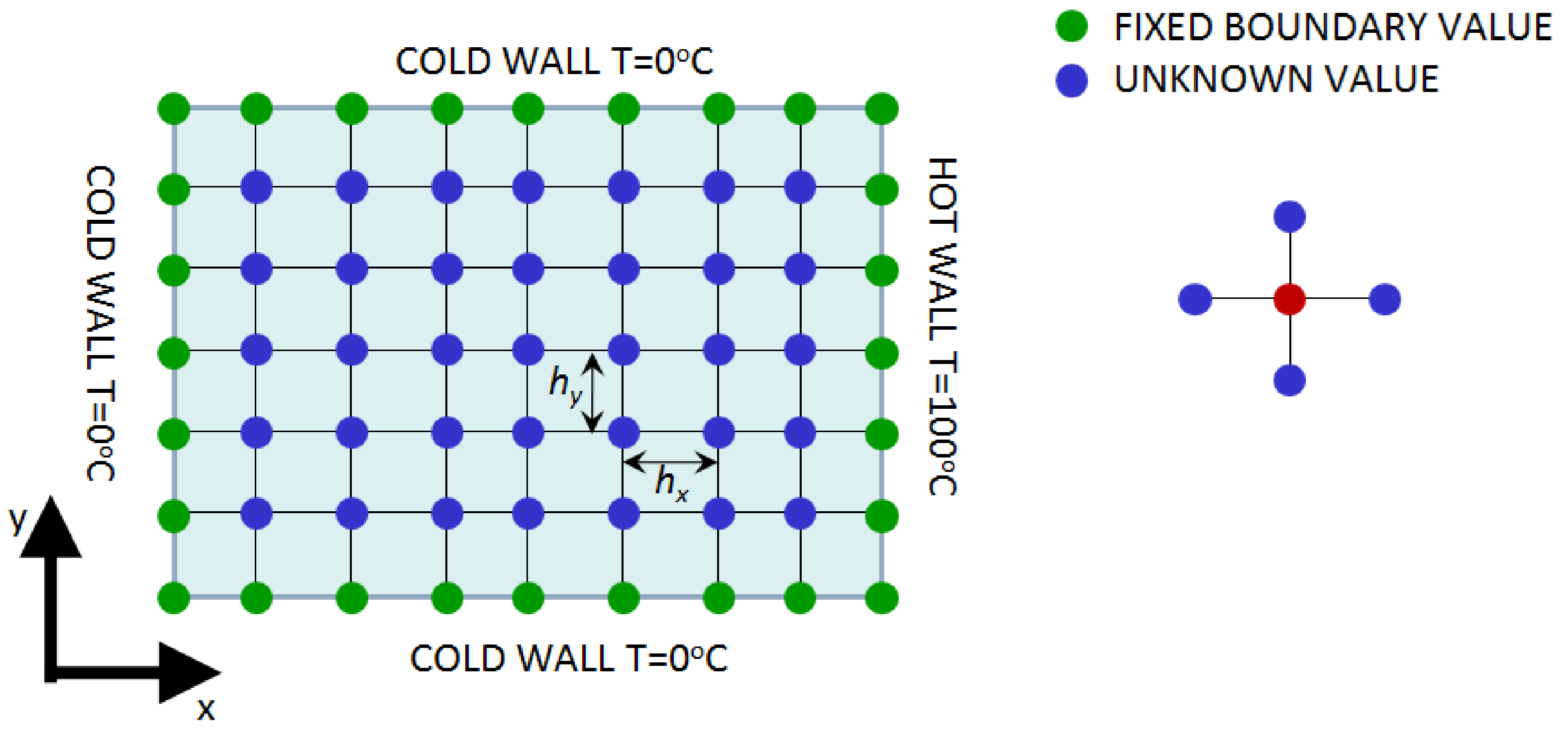


Parallelising Jacobi and Gauss Seidel

UNIVERSITY *of* GREENWICH

How can we parallelise our example?

- 2 Methods:
  - Loop Parallelism: Parallelise loops of our algorithm (OpenMP)
  - Domain decomposition: Break up our problem into smaller chunks then solve them (MPI)



What is loop parallelism?

This was briefly covered in your Week 4 lectures, but now we can get into more details.

Loop parellism is where instead of breaking up your problem into chunks you break up each iterative loop into chunks

Example: populate an array with the first 8 squares

```
For i = 1 to 8
 b[i]=i*i
Next i
```

This is an easy loop to parallelise as no loop depends on the other.

Loop Parallelism Example

Serial:

1st Loop: b[1] = 1*1

2nd Loop: b[2] = 2*2

Etc.

Parallel (2 threads, equal size chunks)

Thread 1

1st Loop: b[1] = 1*1

2nd Loop: b[2] = 2*2

etc.

Thread 2

1st Loop: b[5] = 5*5

2nd Loop: b[6] = 6*6

etc.

Can do the same instructions in half the time as loop is performed at the same time!

Pit falls of Loop parallelism

Loop parallelism is great for some tasks but for others you need to be smarter

Example: sum the first 8 squares

```
For i = 1 to 8
 b=b+i*i
Next I
```

Adding to a variable modified every loop complicate things.

Loop Parallelism Example

Serial:

1st Loop: b=b + 1*1

2nd Loop: b= b+ 2*2

Etc.

Parallel (2 threads, equal size chunks)

Thread 1

1st Loop: b=b + 1*1

2nd Loop: b=b + 2*2

etc.

Thread 2

1st Loop: b = b + 5*5

2nd Loop: b = b+ 6*6

etc.

Clash! If both these execute at the same time your answer will be upredictable

How can we fix this?

Private variables!

Thread 1	Thread 2
1st Loop: b_priv=b_priv + 1*1	1st Loop: b_priv = b_priv + 5*5
2nd Loop: b_priv = b_priv + 2*2	2nd Loop: b_priv = b_priv + 6*6
etc.	etc.
b_tot=b_tot+b_priv	wait
	b_tot=b_tot+b_priv

Due to having to wait for the code to add up we don’t quite get a 2x speedup but for larger numbers this would be quicker.

Note: if you declared b to be a summative reduction variable that would also work, but this is easier to explain

Are there some problems we can’t use loop parallelism for?

Yes! You need “loop independence” to truly parallelise a loop. There are some calculations where you need the solution from your previous loop.

Example: Fibonacci Numbers

The Fibonacci sequence is a sequence of numbers where each number is the sum of the previous two numbers:

1 1 2 3 5 8 13 21 34 55….

This can be generated using a simple loop:

```
f[1] = 1

f[2] = 1

for i = 3 to 8

f[i] = f[i-1]+f[i-2]

next i
```

Parallelising the Fibonacci Sequence

Serial:

1st Loop: f[3]=f[2]+f[1]

2nd Loop: f[4]=f[3]+f[2]

Etc.

Lets try to parallelise like we have before:

Wait! The first thread won't have calculated these numbers yet!

Thread 1	Thread 2
1st Loop: f[3]=f[2]+f[1]	1st Loop: f[6]=f[5]+f[4]
2nd Loop: f[4]=f[3]+f[2]	2nd Loop: f[7]=f[6]+f[5]
etc.	etc.

There is no way to use loop parallelism to parallelise this algorithm!

Loop Parallelism and loop independence

So to loop parallelise some code we need to ensure loop independence

Some algorithms just need small modifications

However some you can’t parallelise.

For Jacobi

The 1D Jacobi algorithm can be written as

```
While not converged
For i =1 to n
  T_new[i] = (T_old[i-1]+T_old[i+1])/2
  T_old[i] = T_new[i]
  Next i
  End while
```

The inner loop is loop independent! So easy to parallelise!

Now we consider only the inner loop

Serial Jacobi

Serial:

1st Loop: T_new[1] =(T_old[0]+T_old[2])/2

T_old[1]=T_new[1]

2nd Loop: T_new[2] =(T_old[1]+T_old[3])/2

T_old[2]=T_new[2]

Etc.

2 Thread Parallel Jacobi (assuming n=10)

Parallel

Thread 1

```
1st Loop: T_new[1] =(T_old[0]+T_old[2])/2
          T_old[1]=T_new[1]
2nd Loop: T_new[2] =(T_old[1]+T_old[3])/2
          T_old[2]=T_new[2]
```

etc.

Thread 2

```
1st Loop: T_new[6] =(T_old[5]+T_old[7])/2
          T_old[6]=T_new[6]
2nd Loop: T_new[7] =(T_old[6]+T_old[8])/2
          T_old[7]=T_new[7]
```

etc.

What about the residual?

There is no point running a Jacobi algorithm if you don't know if you have a converged solution so we need to calculate the residual:

```
While residual_max > 0.0001

For i =1 to n
  T_new[i] =(T_old[i-1]+T_old[i+1])/2
  residual =  abs(T_new[i] –T_old[i])
  T_old[i] = T_new[i]
  residual_max = max(residual_max, residual)
  Next I

End while
```

We've lost our loop independence, but like our previous example we can simply use a private variable to store the max per thread, then calculate the total maximum.

Or use a reduction variable!

Now for Gauss Seidel

```
While not converged
For i =1 to n
  T [i] = (T [i-1]+T [i+1])/2
  Next I
  End while
```

A much simpler algorithm on the face of it

But is it loop independent?

Lets take a look…

Gauss Seidel

Serial:

1st Loop: T1] =(T [0]+T [2])/2,

2nd Loop: T [2] = (T[1]+ T [3])/2,

Etc.

Parallel

Thread 1

1st Loop: T [1] =(T [0]+T [2])/2

2nd Loop: T [2] =(T [1]+T [3])/2

etc.

Thread 2

1st Loop: T [6] =(T [5]+T [7])/2

2nd Loop: T [7] =(T [6]+T [8])/2

etc.

Wait! For Gauss Seidel, this should be T_new, at this stage T_new hasn't been calculated yet

We will still converge on a solution but this is no longer strictly a Gauss Seidel Method! It will also take more iterations of the outer loop to converge! But the speedup from parallelising the code will make up for it for large problems.

Conclusions

Loop independence is vital to use loop parallelism

However when we don’t have loop independence sometimes we can slightly modify an algorithm to get it back.

Some problems however we can’t do this and they cannot be parallelised using loop parallelism.

Gauss Seidel is strictly not parallelisable using loop parallelism.

However we can parallelise a quasi-Gauss Seidel method that will still get us a solution quicker for large problems.