CHIRAG JAIN

HARSHVARDHAN AGRAWAL

RED BLACK METHOD

PSEUDO-CODE:-

While(Value has not converged)

{ If thread\_id==0

Start from i=1 to chunksize

Start from j=1 to dimension-1

If(i+j%2==0)

{

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

}

Else If thread\_id==NUM\_THREADS -1

Start from i=offset to dimension-1

Start from j=1 to dimension-1

If(i+j%2==0)

{

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

}

else

Start from i=offset to offset + chunksize

Start from j=1 to dimension-1

If(i+j%2==0)

{

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

}

Barrier

If thread\_id==0

Start from i=1 to chunksize

Start from j=1 to dimension-1

If(i+j%2==1)

{

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

}

Else If thread\_id==NUM\_THREADS -1

Start from i=offset to dimension-1

Start from j=1 to dimension-1

If(i+j%2==1)

{

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

}

else

Start from i=offset to offset + chunksize

Start from j=1 to dimension-1

If(i+j%2==1)

{

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

}

Acquire\_lock

Final\_diff+=diff;

Release lock

Barrier

Thread 0 updates old grid

If (final\_diff/num elements)< tolerance

stop

}

* In the case of the RED BLACK method to solve the equation, It consists of alternating red and black nodes which are determined by odd and even indexes ie (i+j)%2==0 or (i+j)%2==1;
* In the code, a new grid called grid\_5 is created which keeps track of the value of the grid in the previous iteration and is then used to calculate the new values of the grid elements.
* Since we have to ensure that the boundary rows and columns are neglected in the calculation, conditions are set into place to ensure that the first thread excludes the first row and the last thread excludes the last row. To exclude the first and last columns, j runs from 1 to the second last column of the grid.
* Here every thread first calculates the red nodes that belong to its data chunk and updates the value of grid\_5 and along with increasing a local difference value that is calculated by finding the absolute value of the new value of the element subtracted by the value in the previous iteration.
* All the threads get synchronized at a barrier and the proceed to calculate the black nodes associated with its chunk of data.
* All the threads then get synchronized at a barrier and then race to achieve a lock on the shared variable called final\_diff which is the summation of all the diff values from all the threads.
* At this point, the first thread that achieves the lock increases the iteration value and sets the flag value to 0 so no other thread can increase the iterator value for the same iteration.
* The threads then get synchronized after this point and then thread 0 overwrites the grid\_5 with the new value of grid\_3 which is then used as the old grid in the next iteration.
* Once this is done the final difference is divided by the number of elements are checked with the tolerance.
* If the value is less than the tolerance, the threads all exit otherwise they move to the next iteration.

JACOBIAN METHOD

PSEUDO-CODE:-

While(Value has not converged)

{ If thread\_id==0

Start from i=1 to chunksize

Start from j=1 to dimension-1

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

Else If thread\_id==NUM\_THREADS -1

Start from i=offset to dimension-1

Start from j=1 to dimension-1

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

else

Start from i=offset to offset + chunksize

Start from j=1 to dimension-1

Element=0.20(old self value + old element value above it + old element value below it + old element value to the right + old element value to the left);

Diff+=element new-element old;

Barrier

Acquire\_lock

Final\_diff+=diff;

Release lock

Barrier

Thread 0 updates old grid

If (final\_diff/num elements)< tolerance

stop

}

* In the case of the jacobian method to solve the equation, It differs from the Gauss Sidel method in the way that each element of the grid Is calculated on the basis of the previous values of itself and its neighboring elements.
* In the code, a new grid called grid\_4 is created which keeps track of the value of the grid in the previous iteration and is then used to calculate the new values of the grid elements.
* Here the code is parallelized by distributing the total number of rows into the total number of threads.
* Since we have to ensure that the boundary rows and columns are neglected in the calculation, conditions are set into place to ensure that the first thread excludes the first row and the last thread excludes the last row. To exclude the first and last columns, j runs from 1 to the second last column of the grid.
* Every Thread then calculates the new values of the elements that belong to its allotted chunk of data along with increasing a local difference value that is calculated by finding the absolute value of the new value of the element subtracted by the value in the previous iteration.
* All the threads then get synchronized at a barrier and then race to achieve a lock on the shared variable called final\_diff which is the summation of all the diff values from all the threads.
* At this point, the first thread that achieves the lock increases the iteration value and sets the flag value to 0 so no other thread can increase the iterator value for the same iteration.
* The threads then get synchronized after this point and then thread 0 overwrites the grid\_4 with the new value of grid\_2 which is then used as the old grid in the next iteration.
* Once this is done the final difference is divided by the number of elements are checked with the tolerance.
* If the value is less than the tolerance, the threads all exit otherwise they move to the next iteration.

**SPEEDUP CALCULATED**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time Serial | Time Red Black Parallel | Time Jacobian Parallel | Speedup with Red Black | Speedup with Jacobian |
| 0.14 | 0.1 | 0.07 | 1.4 | 2 |
| 0.13 | 0.08 | 0.05 | 1.63 | 2.6 |
| 0.13 | 0.07 | 0.04 | 1.86 | 3.25 |