Course admin stuff

- Coursework 1 is under way
  - 80 people (!) signed up in github
- Where to find it
  - Spec is on github: https://github.com/HPCE/hpce-2016-cw1
  - Submission for this coursework is via blackboard
    - Submission is open

Expectations for coursework

- Coursework is not lab [1]
  - You have to manage when, where, and how long you spend on it
  - 100% coursework does not mean easy [1]

- Long hours are neither sufficient ***nor*** necessary for an A
  - Like anything else: some people are just good at it
  - But, a good correlation between organisation and marks

- You are expected to be reasonably independent
  - This is a masters level course

[1] – Though the earlier parts kind of are.

Working together

- The software community has a tradition of sharing
  - Many open-source projects, some of which you will rely on
  - Lots of forums for discussing problems: stack-overflow, ...
- Approach this work in the same way
  - You may encounter the same problems as other students
  - Discuss solutions with each other, help each other out
  - One-on-one discussions, github issues, whatever
    - https://github.com/HPCE/hpce-2015-cw1/blob/master/background-bugs.md
  - Give credit or thanks if appropriate: be excellent to each other

- But you have to balance co-operation and competition
  - The later courseworks require good ideas and strategies
  - Up to you to protect your IP.

Plagiarism

- All submitted material must be written by you
  - Do not share any code with each other (except within pairs)
- No plagiarism checking software: I just read the code
  - Some similarity of structure is expected, but there are limits
  - Students are amusingly bad at obfuscation
  - You need to be able to explain any code you submit in the oral
  - Suspected plagiarism will be passed to the plagiarism committee

- If necessary you ***may*** use code from third-party sources
  - e.g. open-source projects, samples, stack overflow, ...
  - Origin and extent must be very clearly shown
  - Need to be able to justify (orally) why it was used
  - Should be aware of potential licensing implications

More practical: image processing

- *Gamma correction*: adjust light/dark
  - $p_{[x,y]} = \text{pow}(p_{[x,y]}, \text{gamma})$

γ = 2

γ = 1

γ = 0.5

```
void process_frame(
    float gamma,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
){
    for(unsigned x=   0u; x<width; x++){
        for(unsigned y=0; y<height; y++){
            double fIn = frameIn[y*width + x] * (1.0/256.0);

            double fOut = pow( fIn, gamma );

            frameOut[y*width + x] = (uint8_t)floor(fOut * 256.0);
        }
    }
}
```

```
void process_frame(
    float gamma,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
){
    tbb::parallel_for(0u,   width, [&](unsigned x){
        for(unsigned y=0; y<height; y++){
            double fIn = frameIn[y*width + x] * (1.0/256.0);

            double fOut = pow( fIn, gamma );

            frameOut[y*width + x] = (uint8_t)floor(fOut * 256.0);
        }
    });
}
```

f is a variable, but we let compiler decide on its type

Capture variables by reference (can modify outer variables)

Lambda parameters, just like function parameter.

```
void process_frame(
    float gamma,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
){
    auto f =                                [&](unsigned x){
        for(unsigned y=0; y<height; y++){
            double fIn = frameIn[y*width + x] * (1.0/256.0);

            double fOut = pow( fIn, gamma );

            frameOut[y*width + x] = (uint8_t)floor(fOut * 256.0);
        }
    };

    tbb::parallel_for(0u,   width,
        f
    );
}
```

```
void process_frame(
    float gamma,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
){
    auto f =                          [&](unsigned x){
        for(unsigned y=0; y<height; y++){
            double fIn = frameIn[y*width + x] * (1.0/256.0);

            double fOut = pow( fIn, gamma );

            frameOut[y*width + x] = (uint8_t)floor(fOut * 256.0);
        }
    };

    for(unsigned i=   0u; i<width; i++){
        f(i);
    }
}
```

Quantisation via dithering

- Dithering: cumulative error due to quantisation is tracked

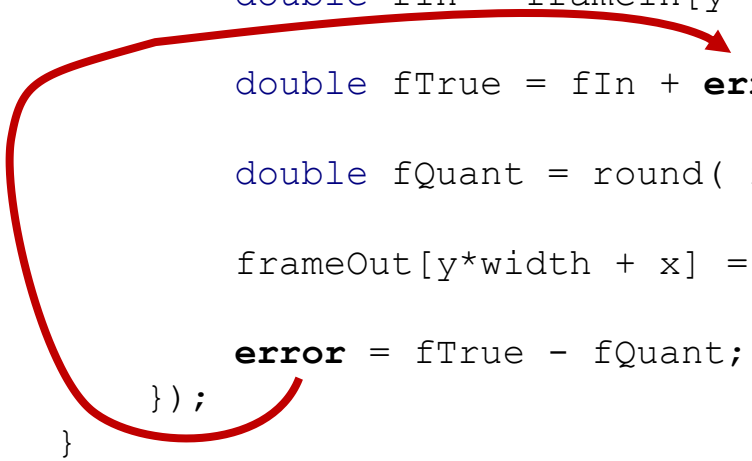
Quantisation via error dithering

```
void process_frame(
    unsigned levels,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
)
{
    for(unsigned x=0; x<width; x++){
        double error=0.0;
        for(unsigned y=0; y<height; y++){
            double fIn = frameIn[y*width + x] * (1.0/255.0);

            double fTrue = fIn + error;

            double fQuant = round( fTrue * levels ) / levels;

            frameOut[y*width + x] = (uint8_t)floor(fQuant * 255.0);

            error = fTrue - fQuant;
        }
    }
}
```

Loop carried dependency through error

Parallelising the inner loop

```
void process_frame(
    unsigned levels,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
)
{
    for(unsigned x=0; x<width; x++){
        double error=0.0;
        tbb::parallel_for(0u, height, [&](unsigned y){
            double fIn = frameIn[y*width + x] * (1.0/255.0);

            double fTrue = fIn + error;

            double fQuant = round( fTrue * levels ) / levels;

            frameOut[y*width + x] = (uint8_t)floor(fQuant * 255.0);

            error = fTrue - fQuant;
        });
    }
}
```



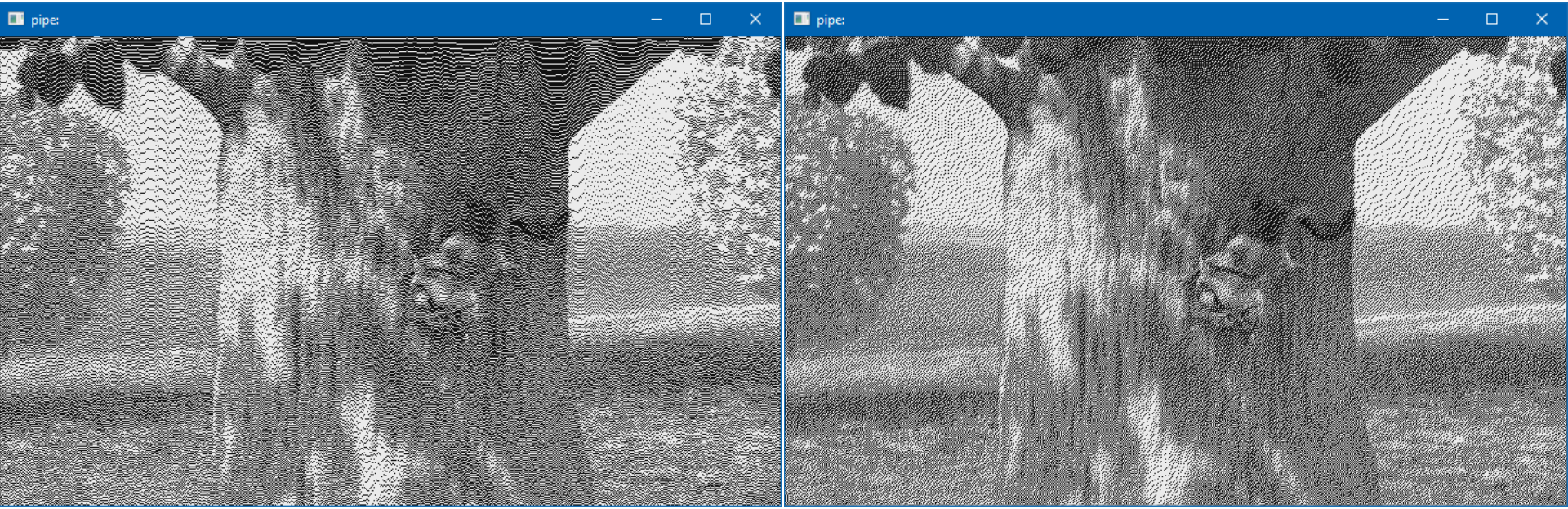
Parallelising the outer loop

```
void process_frame(
    unsigned levels,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
)
{
    tbb::parallel_for(0u, width, [&](unsigned x){
        double error=0.0;
        for(unsigned y=0; y<height; y++){
            double fIn = frameIn[y*width + x] * (1.0/255.0);

            double fTrue = fIn + error;

            double fQuant = round( fTrue * levels ) / levels;

            frameOut[y*width + x] = (uint8_t)floor(fQuant * 255.0);

            error = fTrue - fQuant;
        }
    });
}
```

A solution for the inner loop?

```
void process_frame(
    unsigned levels,
    unsigned width, unsigned height,
    const uint8_t *frameIn,
    uint8_t *frameOut
)
{
    std::vector<double> error(height, 0.0);

    for(unsigned x=0; x<width; x++){
        tbb::parallel_for(0u, height, [&](unsigned y){
            double fIn = frameIn[y*width + x] * (1.0/255.0);
            assert( (fIn>=0) && (fIn<=1.0));

            double fTrue = fIn + error[y];

            double fQuant = round( fTrue * levels ) / levels;

            frameOut[y*width + x] = (uint8_t)floor(fQuant * 255.0);

            error[y] = fTrue - fQuant;
        });
    }
}
```

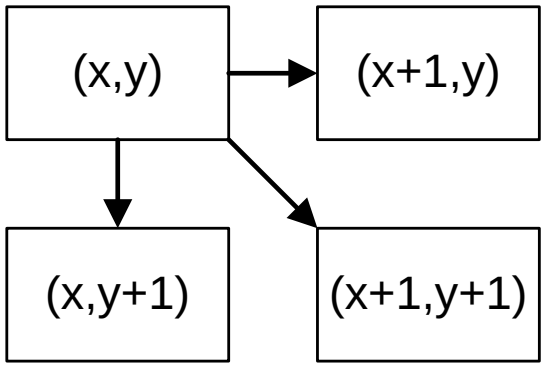
2D error diffusion



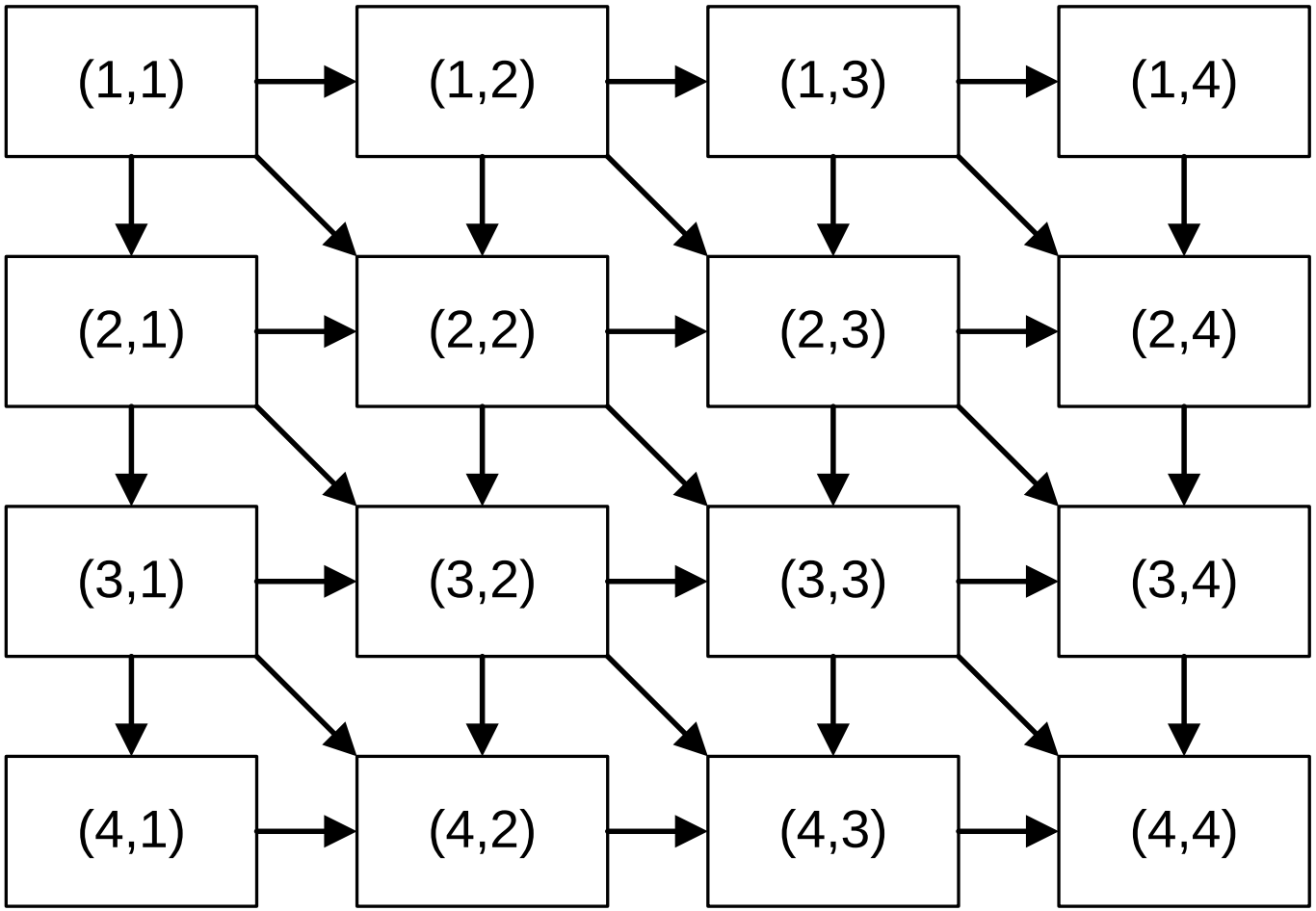
- Attempt to diffuse error both across and down image
- Reduce tendency towards banding effects

```
void process_frame(
    unsigned levels, unsigned width, unsigned height,
    double *frame
)
{
    for(unsigned x=0; x<width-1; x++){
        for(unsigned y=0; y<height-1; y++){
            double fIn = frame[y*width + x];

            double fQuant = round( fIn * levels ) / levels;
            frame[y*width + x] = fQuant;

            double error = fIn - fQuant;
            frame[  y    * width + x+1] += error * 0.4;
            frame[ (y+1) * width + x  ] += error * 0.4;
            frame[ (y+1) * width + x+1] += error * 0.2;
        }
    }
}
```

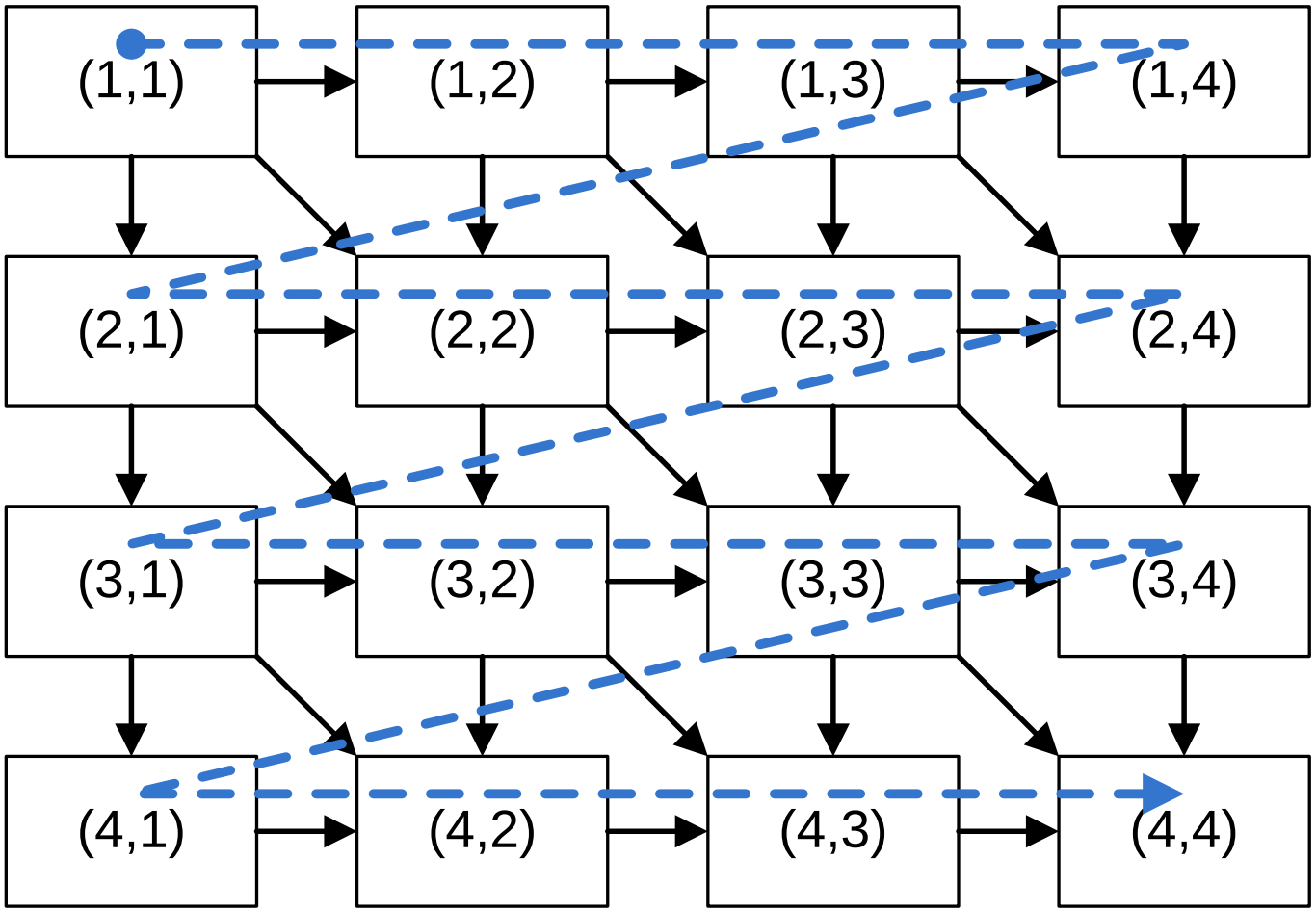
- More difficult loop carried dependency
- Have a write before read dependency
  - Current loop iteration reads from (x,y)
  - Writes (x,y+1), (x+1,y), and (x+1,y+1)
  - Three constraints per node



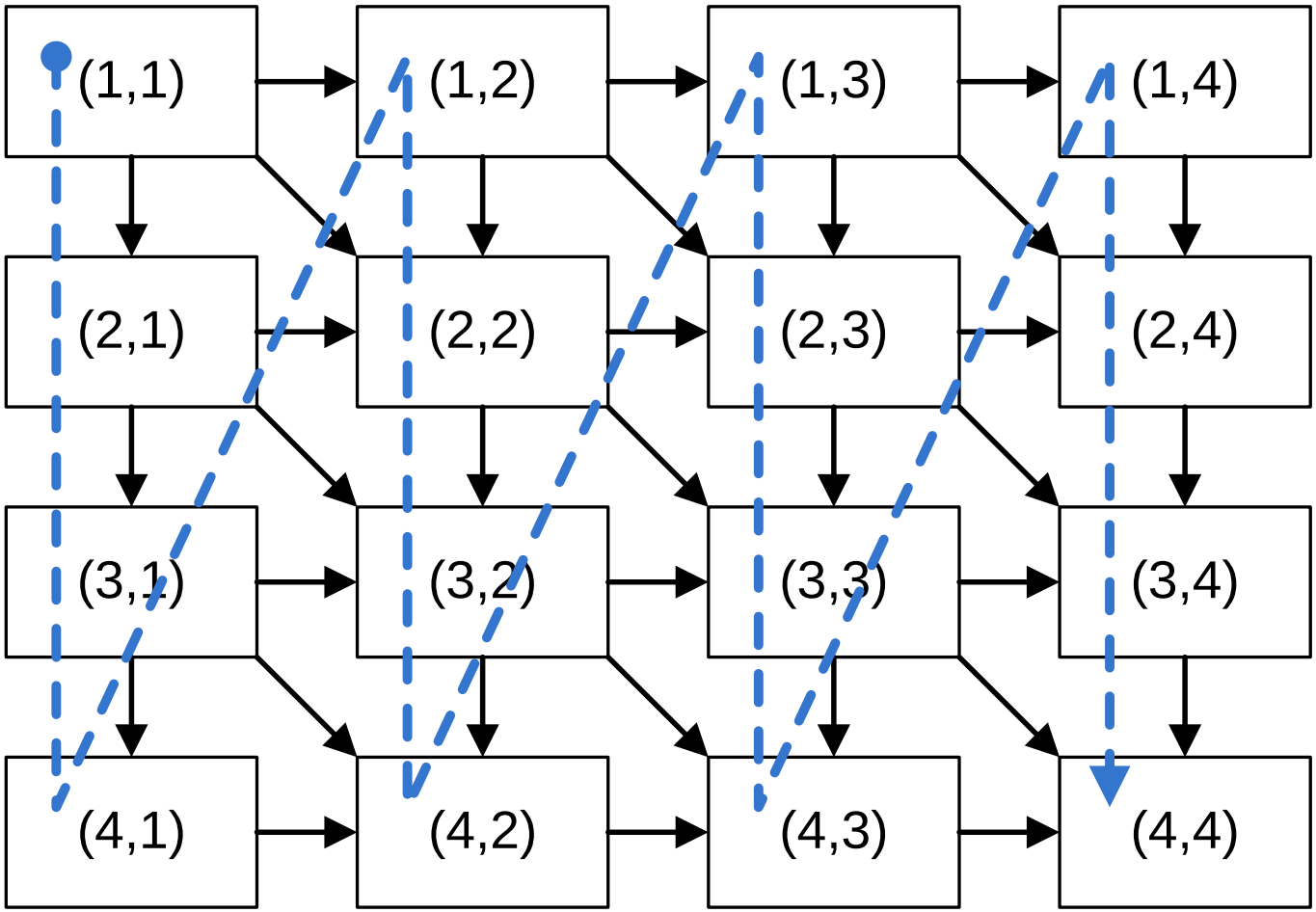
Generalise to the full grid



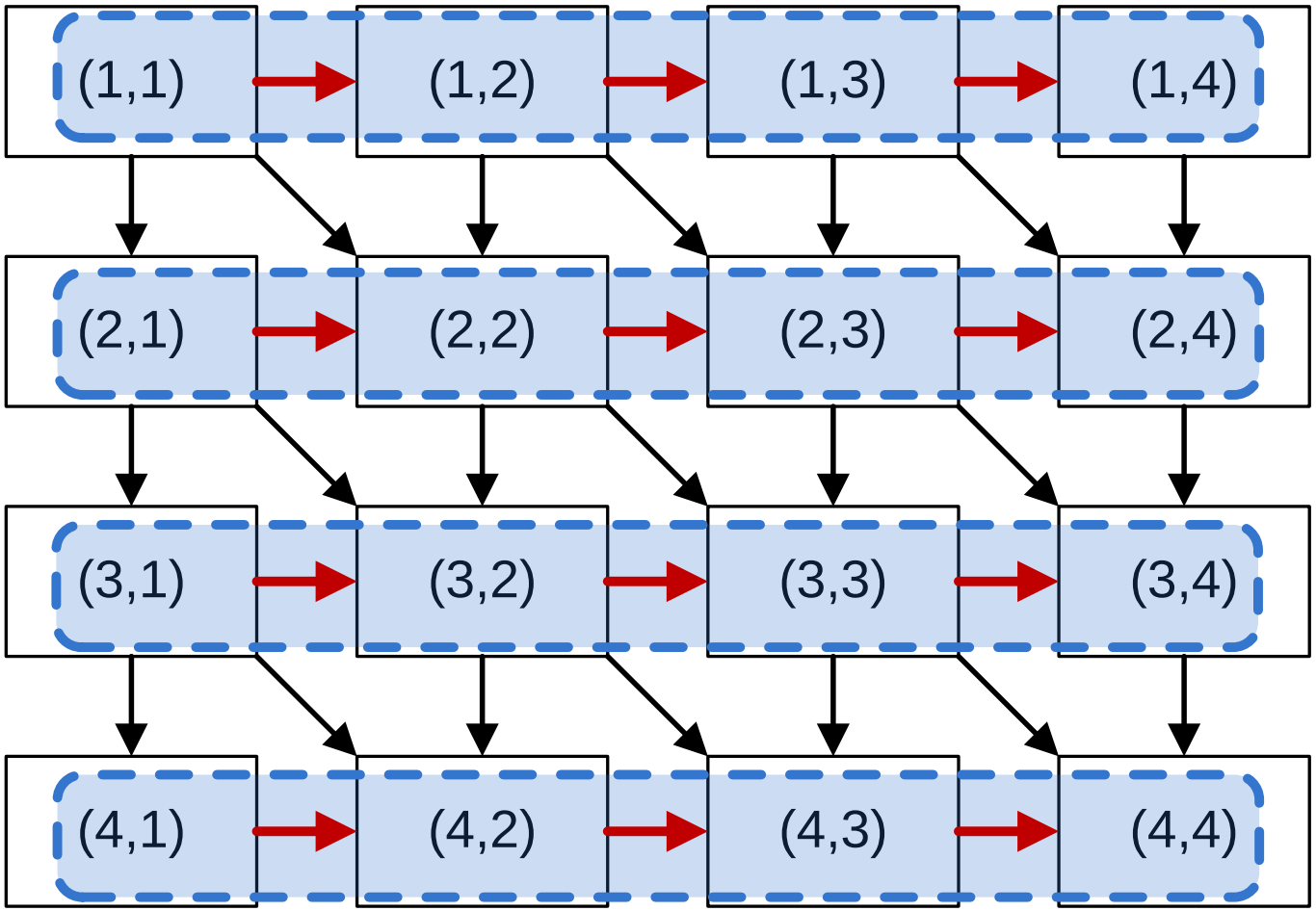
Serial execution: y then x



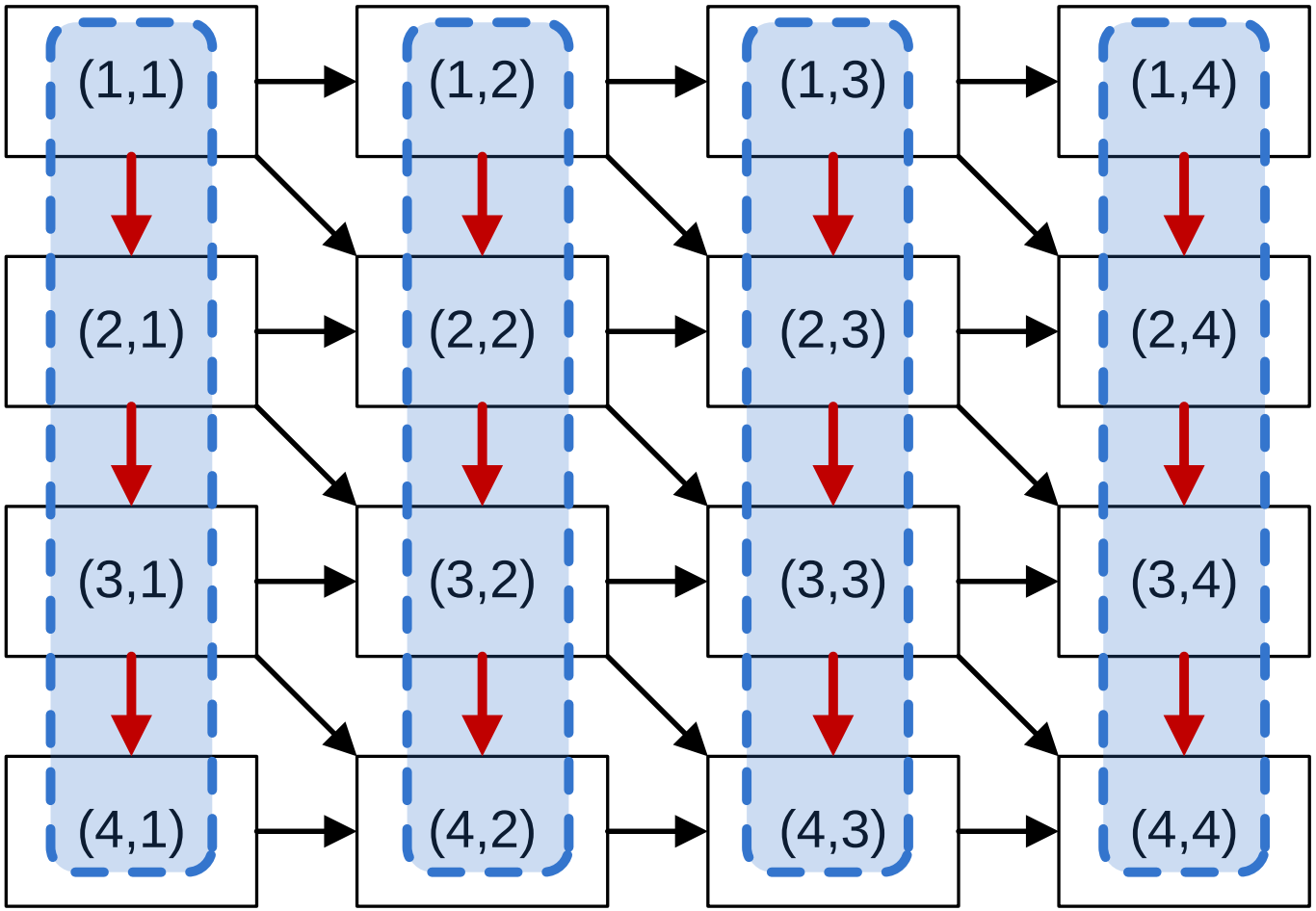
Serial execution: x then y



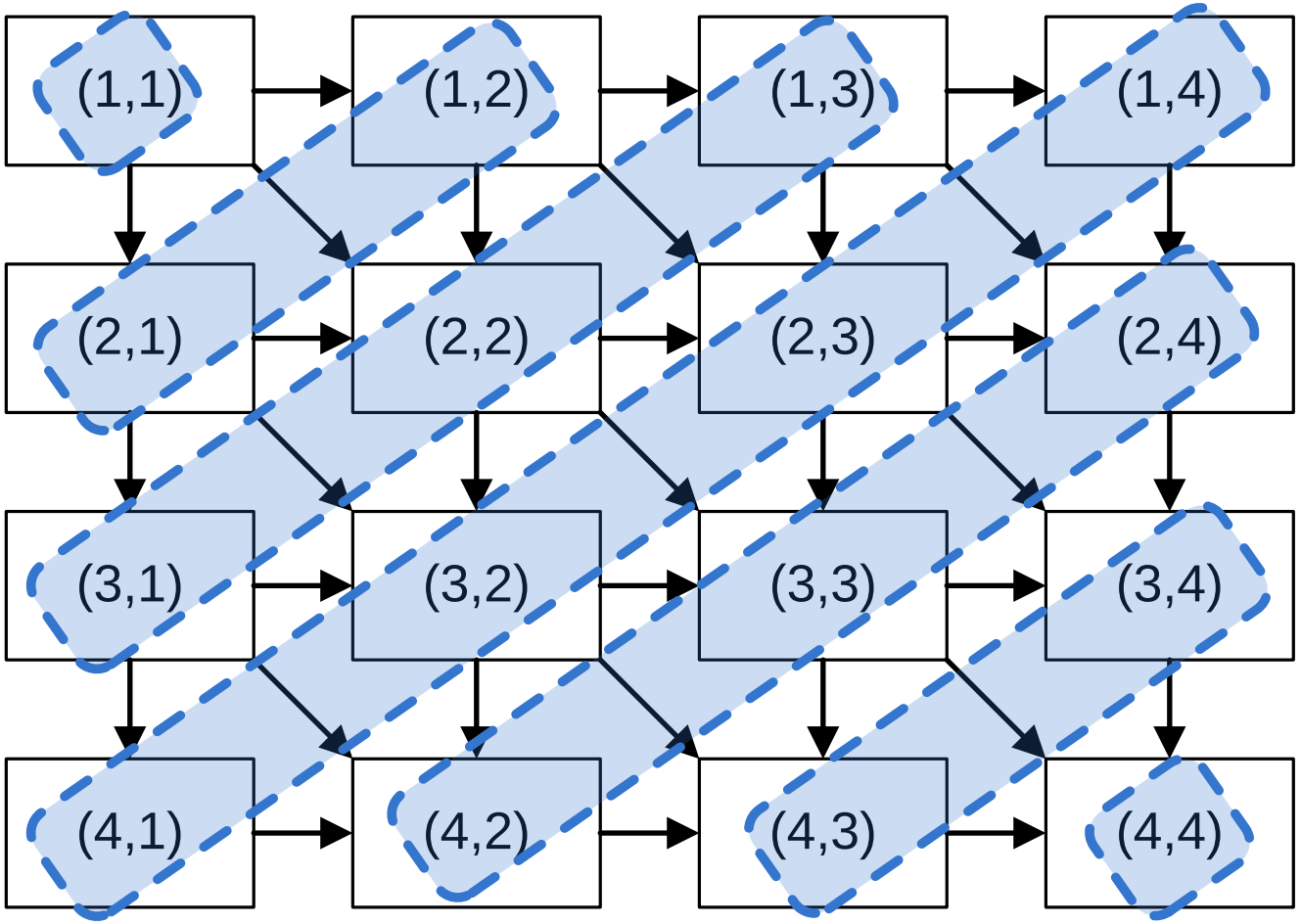
Parallelisation: can’t do it along x



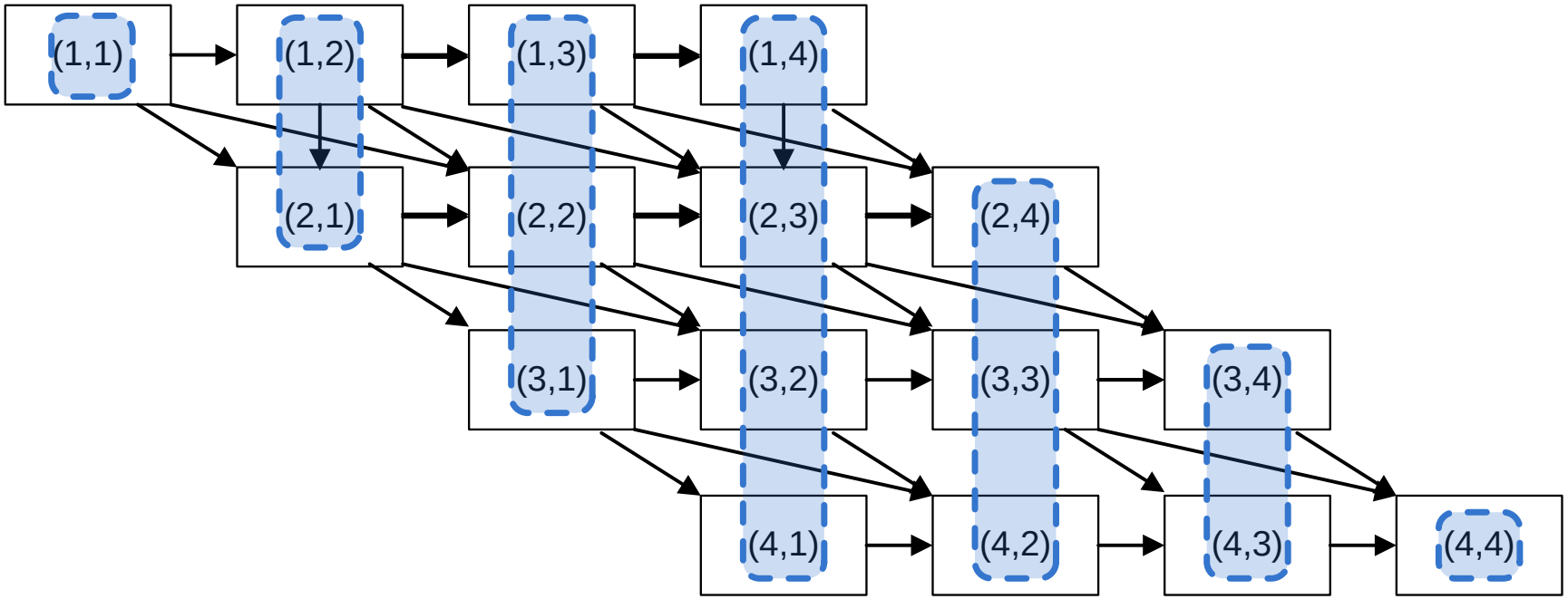
Parallelisation: can’t do it along y



Skewing the loops

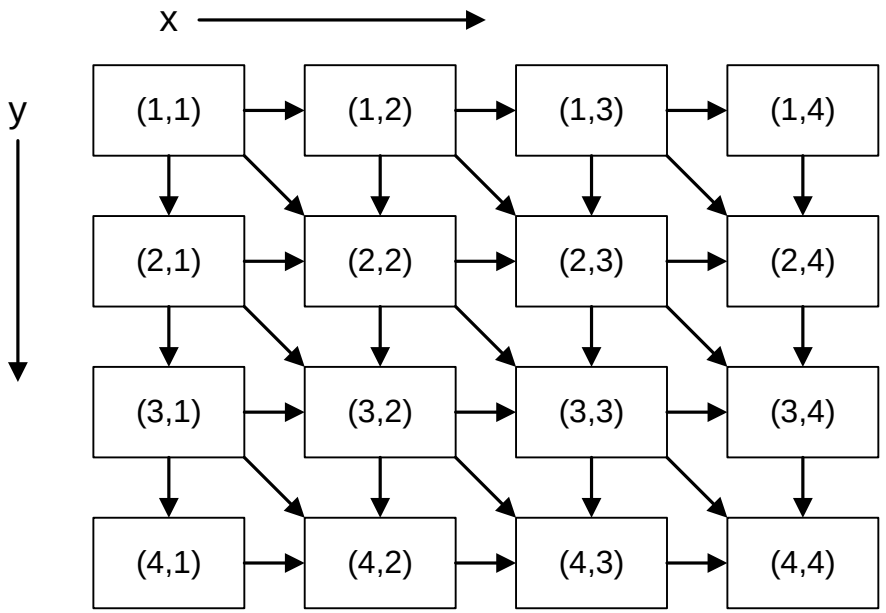


Or viewed another way



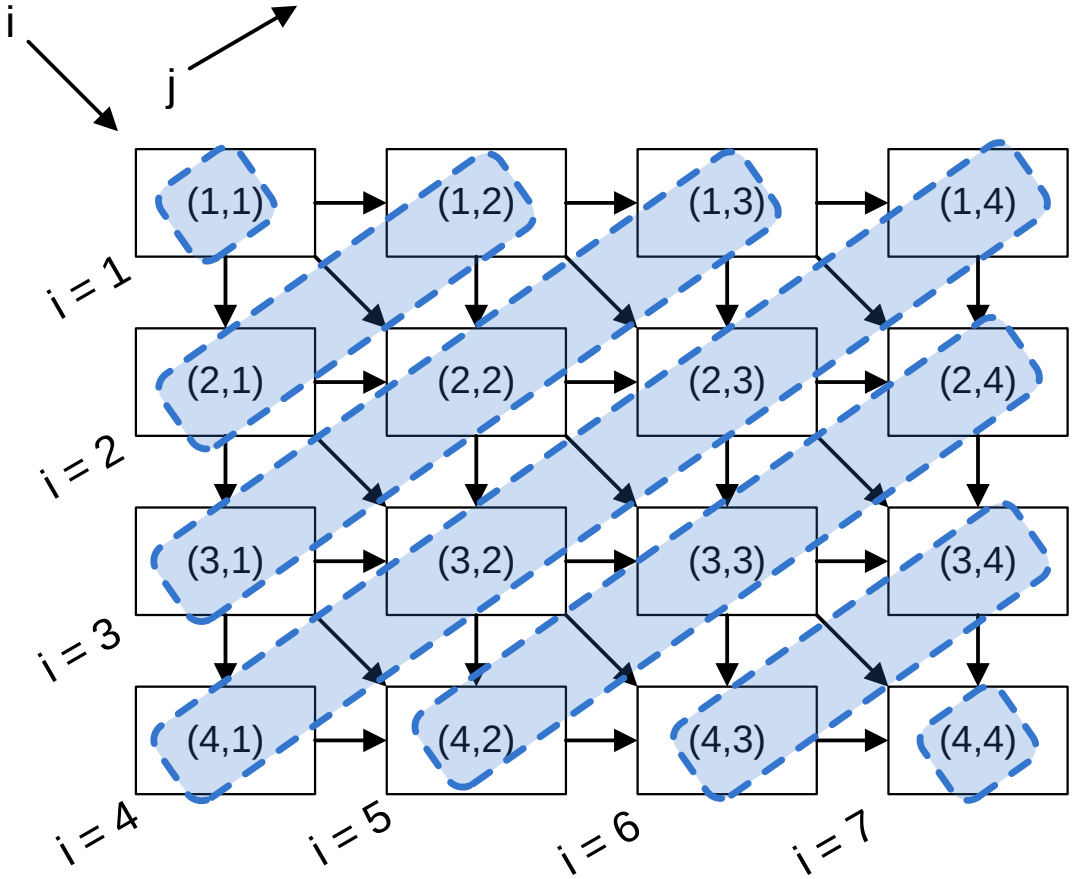
Iteration spaces

- We already have iteration as a primitive
  - for loops : bounded iteration over known range
  - while loops : possibly unbounded iteration (though maybe not)
- Iteration spaces assign unique labels to distinct iterations



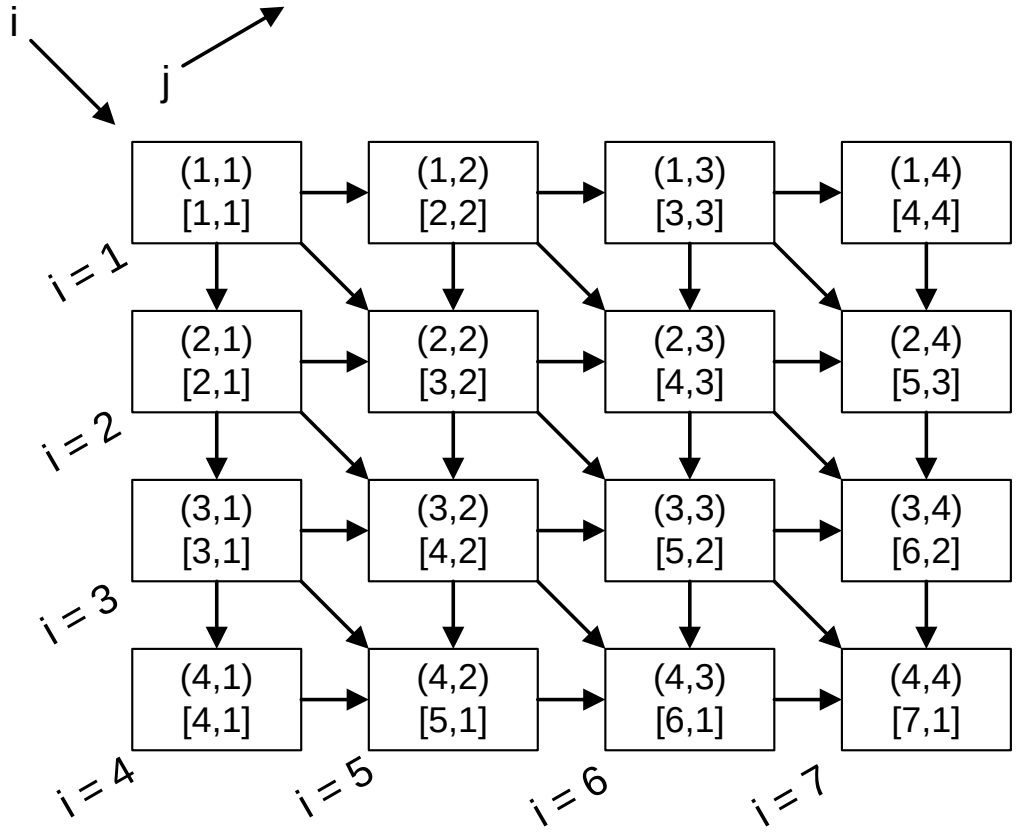
Transforming iteration spaces

- We now want to map (x,y) to a new iteration space [i,j]
  - A different set of loop variables that expose parallel operations



Transforming iteration spaces

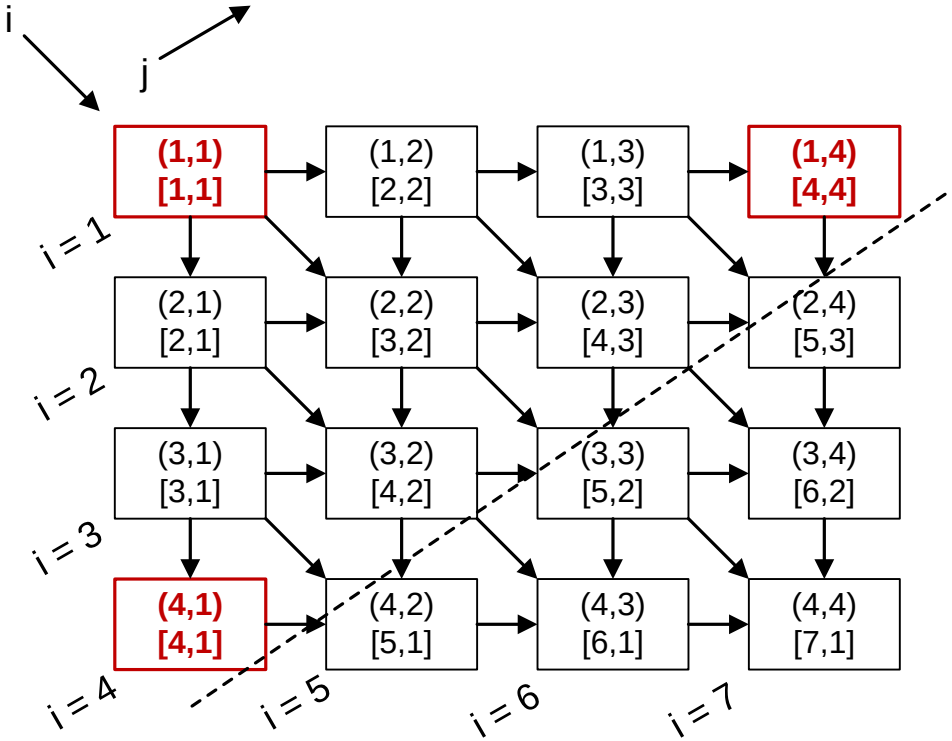
- Mapping must be distinct: all (x,y) go to a different [i,j]



Solving for the equations

- Mapping must be distinct: all (x,y) go to a different [i,j]

```
for(unsigned i=0; i<n-1; i++){
    for(int j=i; j>=0; j--){
        unsigned x=i;
        unsigned y=i-j;
```



Back to the code!

```
void process_frame(unsigned levels,unsigned width,unsigned height,double *frame)
{
    unsigned n=std::min(width,height);

    for(unsigned i=0; i<n-1; i++){
        for(int j=i; j>=0; j--){
            unsigned x=i;
            unsigned y=i-j;

            double fIn = frame[y*width + x];

            double fQuant = round( fIn * levels ) / levels;

            frame[y*width + x] = fQuant;

            double error = fIn - fQuant;
            frame[  y    * width + x+1] += error * 0.4;
            frame[ (y+1) * width + x  ] += error * 0.4;
            frame[ (y+1) * width + x+1] += error * 0.2;
        }
    }
}
```

Note: this only handles *half* the iteration space

Parallel! Correct?

```
void process_frame(unsigned levels,unsigned width,unsigned height,double *frame){
    unsigned n=std::min(width,height);

    for(unsigned i=0; i<n-1; i++){
        tbb::parallel_for(0u, i+1, [&](unsigned rev_j){
            int j=i-(int)rev_j;
            unsigned x=i;
            unsigned y=i-j;

            double fIn = frame[y*width + x];

            double fQuant = round( fIn * levels ) / levels;

            frame[y*width + x] = fQuant;

            double error = fIn - fQuant;
            frame[  y    * width + x+1] += error * 0.4;
            frame[ (y+1) * width + x  ] += error * 0.4;
            frame[ (y+1) * width + x+1] += error * 0.2;
        });
    }
}
```

Note: this only handles *half* the iteration space