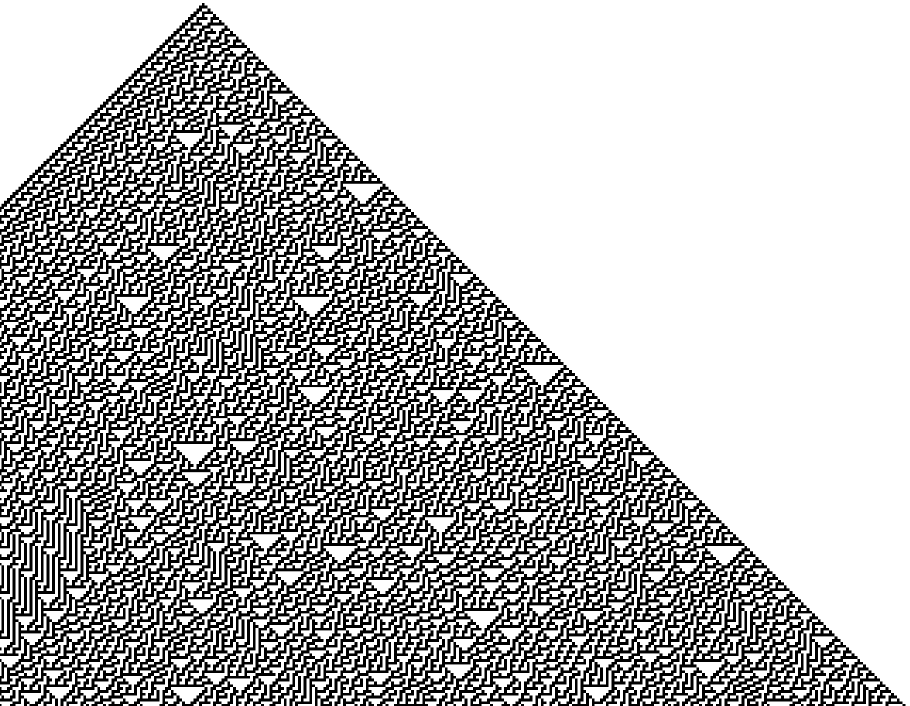
Fall 2012

CSE 633 Parallel Algorithms

Cellular Automata



Nils Wisiol

11/13/12

Simple Automaton: Conway's Game of Life

Simple Automaton: Conway's Game of Life

John H. Conway

Simple Automaton: Conway's Game of Life

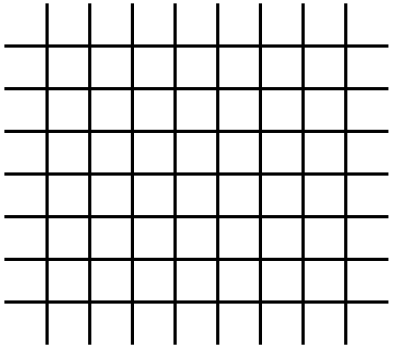
John H. Conway

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

Simple Automaton: Conway's Game of Life

John H. Conway

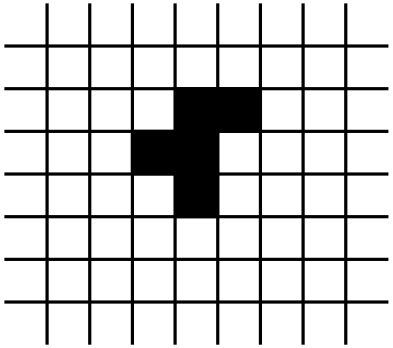
Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).



Simple Automaton: Conway's Game of Life

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

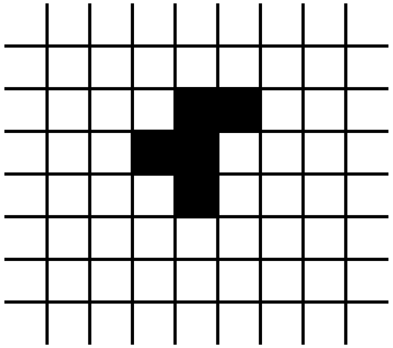
John H. Conway



Simple Automaton: Conway's Game of Life

John H. Conway

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

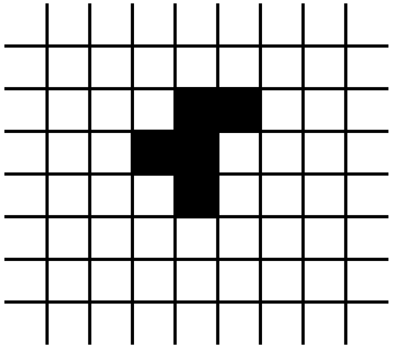


To move on from the current state to the next generation, we update the grid according to the game's rules.

Simple Automaton: Conway's Game of Life

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

John H. Conway



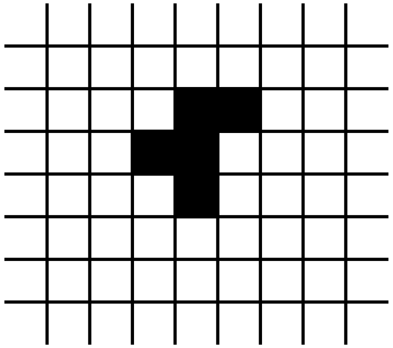
To move on from the current state to the next generation, we update the grid according to the game's rules.

The Rules

Simple Automaton: Conway's Game of Life

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

John H. Conway



To move on from the current state to the next generation, we update the grid according to the game's rules.

The Rules

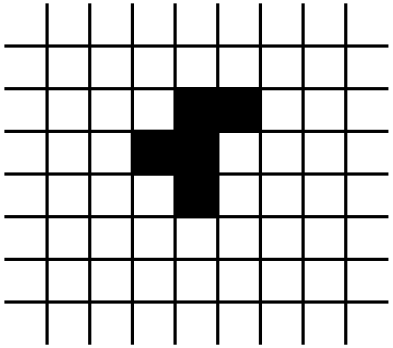
Each *alive* cell,

- stays alive if it has two or three neighbours,
- otherwise it dies.

Simple Automaton: Conway's Game of Life

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

John H. Conway



To move on from the current state to the next generation, we update the grid according to the game's rules.

The Rules

Each *alive* cell,

- stays alive if it has two or three neighbours,
- otherwise it dies.

Any *dead* cell, that has

- exactly three neighbours, becomes alive,
- otherwise it remains dead.

Simple Automaton: Conway's Game of Life

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

John H. Conway

Short:

ules

alive cell,

- stays alive if it has two or three neighbours,
- otherwise it dies.

Any *dead* cell, that has

- exactly three neighbours, becomes alive,
- otherwise it remains dead.

To move on from the current state to the next generation, we update the grid according to the game's rules.

Simple Automaton: Conway's Game of Life

Let's have a two-dimensional grid with cells, and each of the cells can be dead (white) or alive (black).

John H. Conway

Short:

For neighbourhood count i,
cell $\leftarrow$ alive ? $i = 2$ or $i = 3$: $i = 3$;

To move on from the current state to the next generation, we update the grid according to the game's rules.

ules

n *alive* cell,

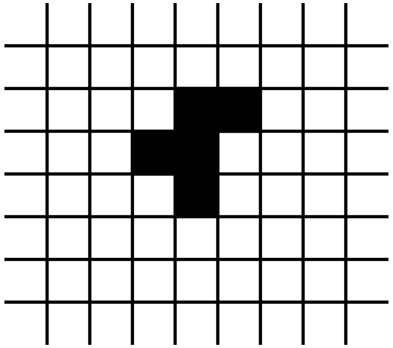
- stays alive if it has two or three neighbours,
- otherwise it dies.

Any *dead* cell, that has

- exactly three neighbours, becomes alive,
- otherwise it remains dead.

Single Core Implementation

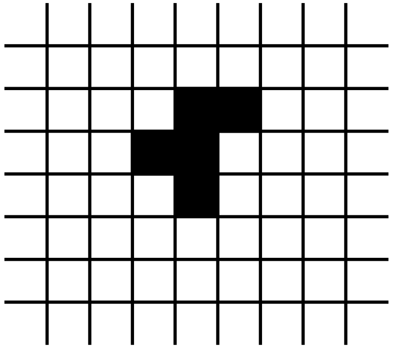
The cell data is maintained in a two-dimensional boolean array, where the first index is the row and the second index gives the column of any cell.



Single Core Implementation

The cell data is maintained in a two-dimensional boolean array, where the first index is the row and the second index gives the column of any cell.

```
bool **world, **buffer;
```

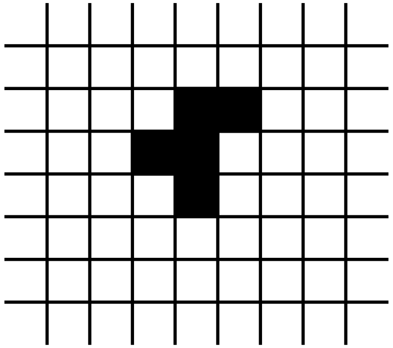


Single Core Implementation

The cell data is maintained in a two-dimensional boolean array, where the first index is the row and the second index gives the column of any cell.

```
bool **world, **buffer;
```

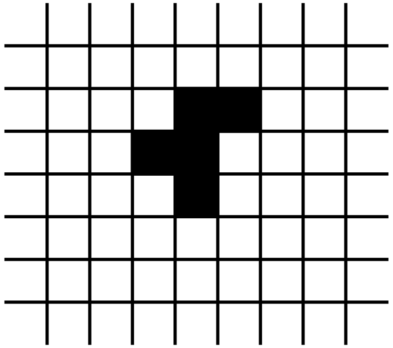
`world` represents the current state, `buffer` the state that is currently calculated.



Single Core Implementation

The cell data is maintained in a two-dimensional boolean array, where the first index is the row and the second index gives the column of any cell.

```
bool **world, **buffer;
```

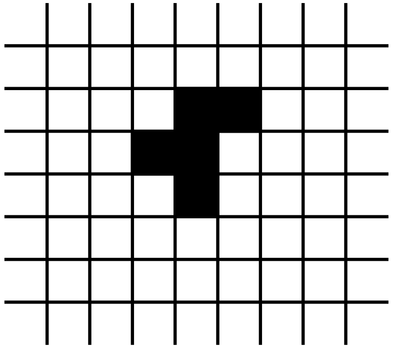


`world` represents the current state, `buffer` the state that is currently calculated.

```
step()
  for each row $j$
    for each row $i$
      c $\leftarrow$ countN(j,i)
      buffer[j][i] $\leftarrow$ rule(c)
  swap(world, buffer)
```

Single Core Implementation

The cell data is maintained in a two-dimensional boolean array, where the first index is the row and the second index gives the column of any cell.



```
bool **world, **buffer;
```

`world` represents the current state, `buffer` the state that is currently calculated.

countN() calculates the count of alive neighbours of the cell in row j and column i, rule() implements Conway's Game Of Life Rule.

```
step()
  for each row $j$
    for each row $i$
      c $\leftarrow$ countN(j,i)
      buffer[j][i] $\leftarrow$ rule(c)
  swap(world, buffer)
```

Single Core Input Size Benchmark

The analysis of this simple simulation algorithm shows that for a board with n cells, the runtime for a fixed number of generations is $O(n)$.

Single Core Input Size Benchmark

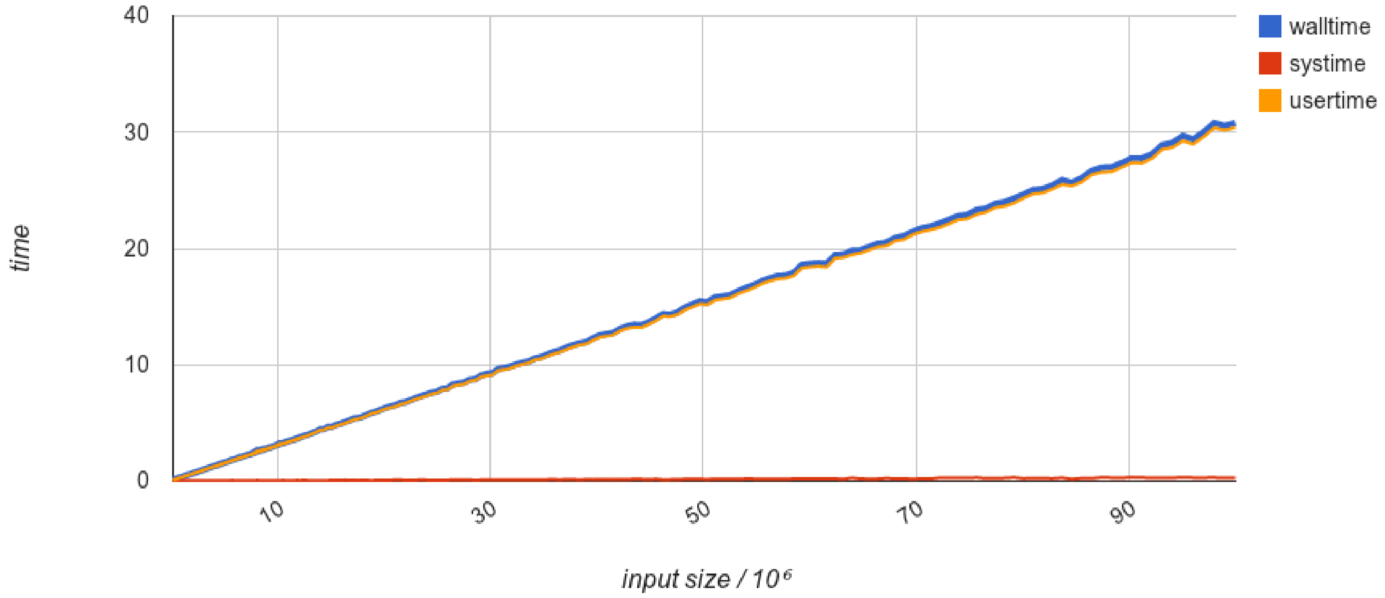
The analysis of this simple simulation algorithm shows that for a board with n cells, the runtime for a fixed number of generations is $O(n)$.

The measured times on a Core i5 support this theoretical result.

Single Core Input Size Benchmark

The analysis of this simple simulation algorithm shows that for a board with n cells, the runtime for a fixed number of generations is $O(n)$.

The measured times on a Core i5 support this theoretical result.



OpenMP Implementation

When storing `world` and `buffer` in shared memory, parallel implementation is straightforward.

OpenMP Implementation

When storing `world` and `buffer` in shared memory, parallel implementation is straightforward.

Using `#pragma omp for` to execute loop in parallel.

OpenMP Implementation

When storing `world` and `buffer` in shared memory, parallel implementation is straightforward.

Using `#pragma omp for` to execute loop in parallel.

```
void step() {
#pragma omp for
  for (int j = 0; j < HEIGHT; j++) {
    for (int i = 0; i < WIDTH; i++) {
      int c = countN(i, j);
      buffer[j][i] = world[j][i] ? (c == 2 || c == 3) : c == 3;
    }
  }
}
```

OpenMP Implementation

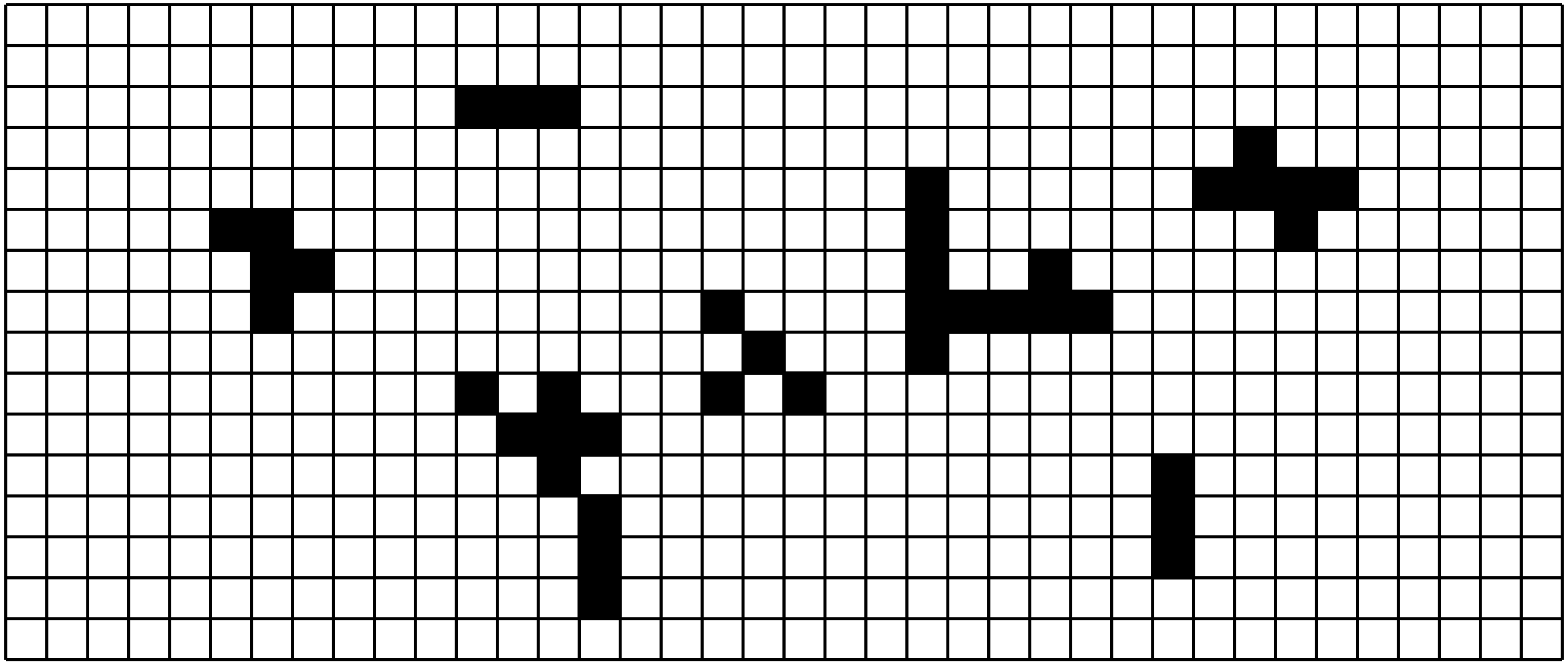
When storing world and buffer in shared memory, parallel implementation is straightforward.

Using #pragma omp for to execute loop in paral

```
void step()
#pragma omp
  for (int j
    for (int
      int c = countN(i, j);
      buffer[j][i] = world[j][i] ? (c == 2 || c == 3) : c == 3;
    }
  }
}
```

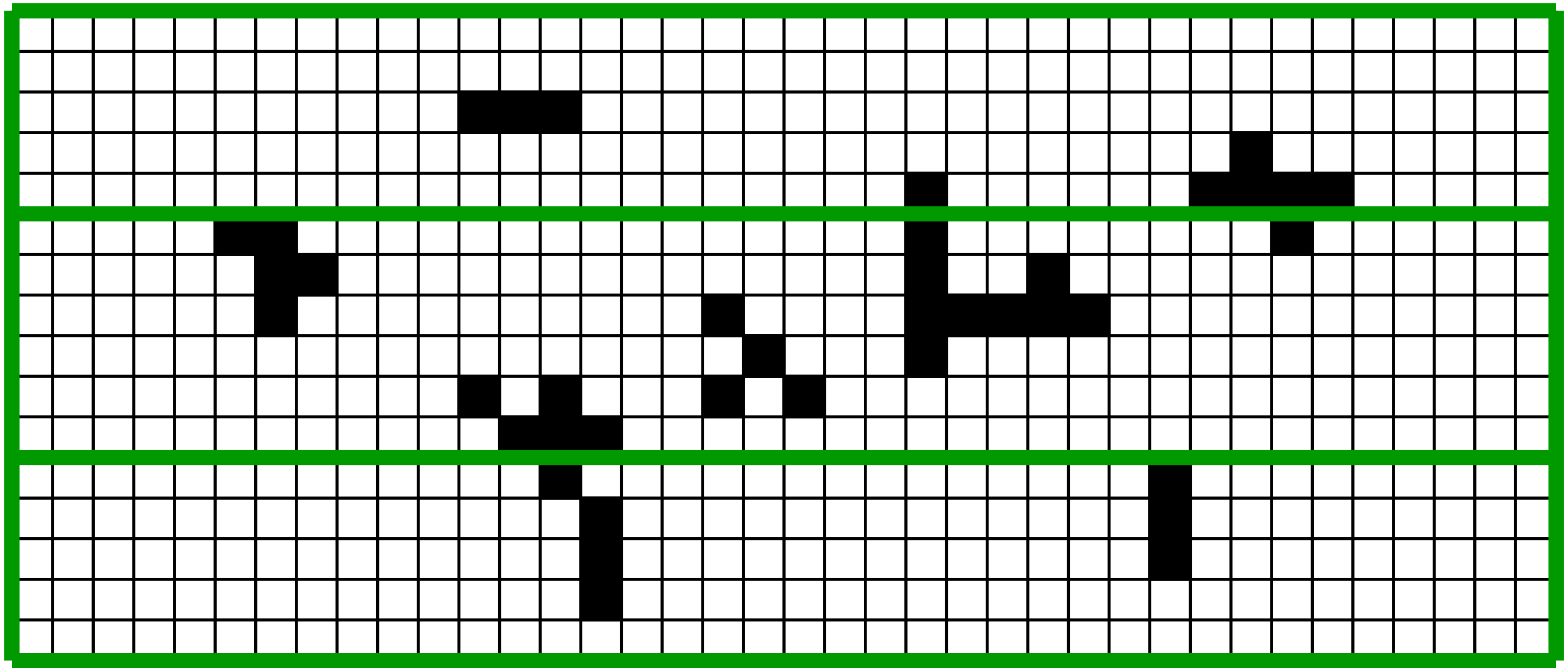
✓ speedup of 30.4 on the 32 core machine

OpenMPI Implementation



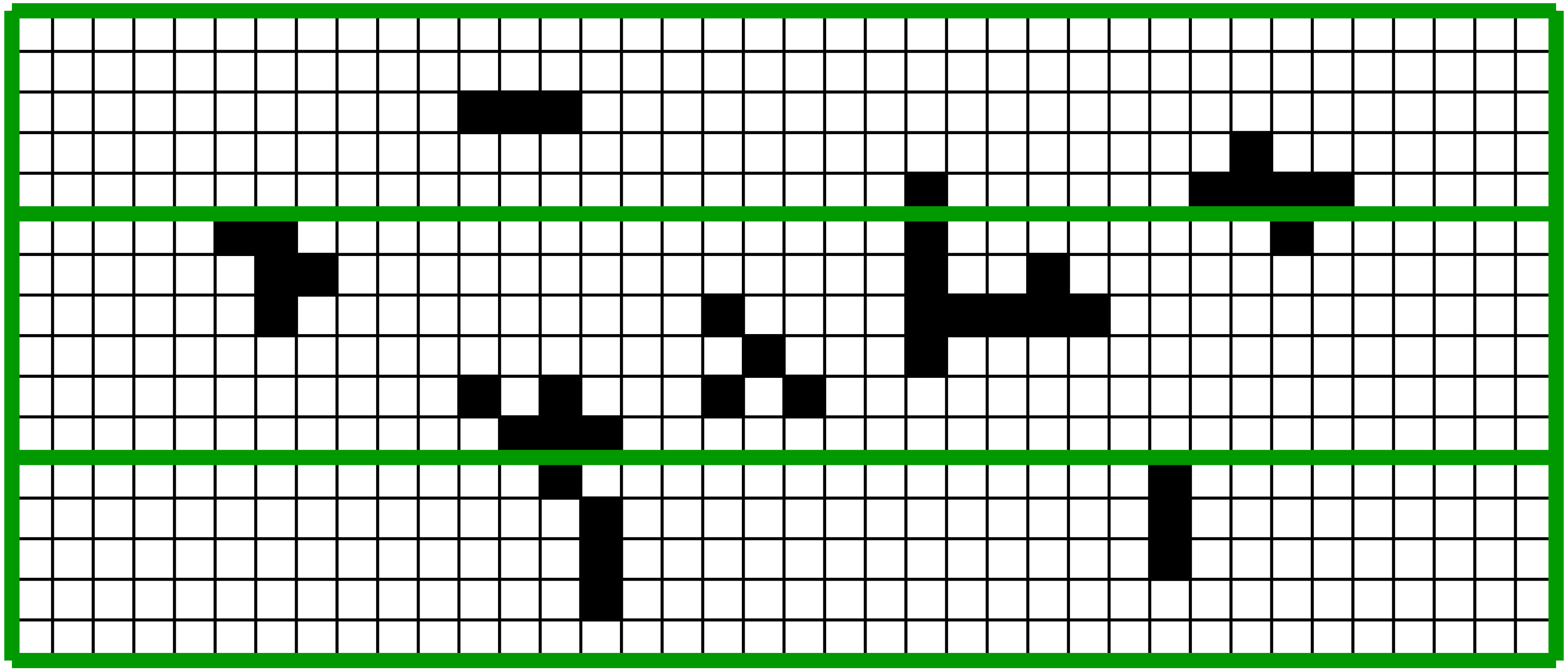
OpenMPI Implementation

- devide the world in t pieces, each for one process



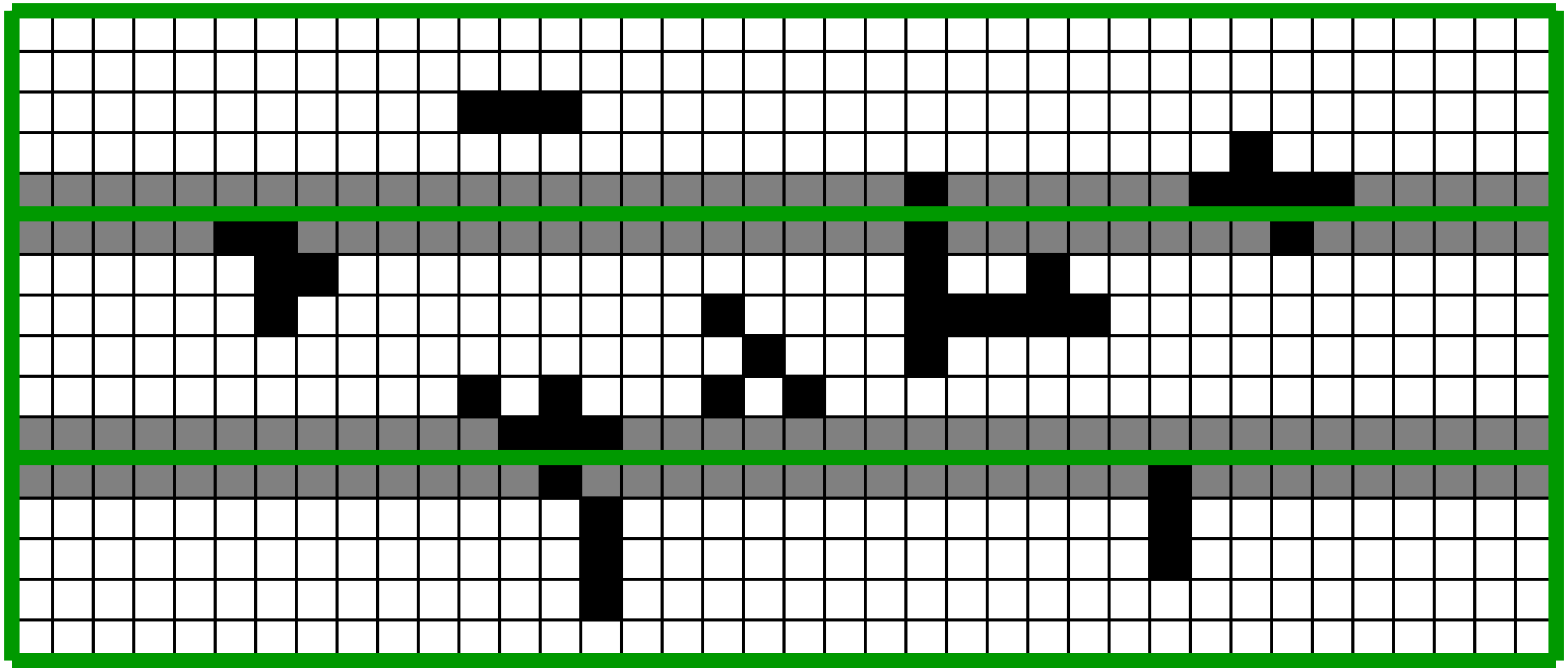
OpenMPI Implementation

- devide the world in t pieces, each for one process
- calculate cell by cell, generation by generation



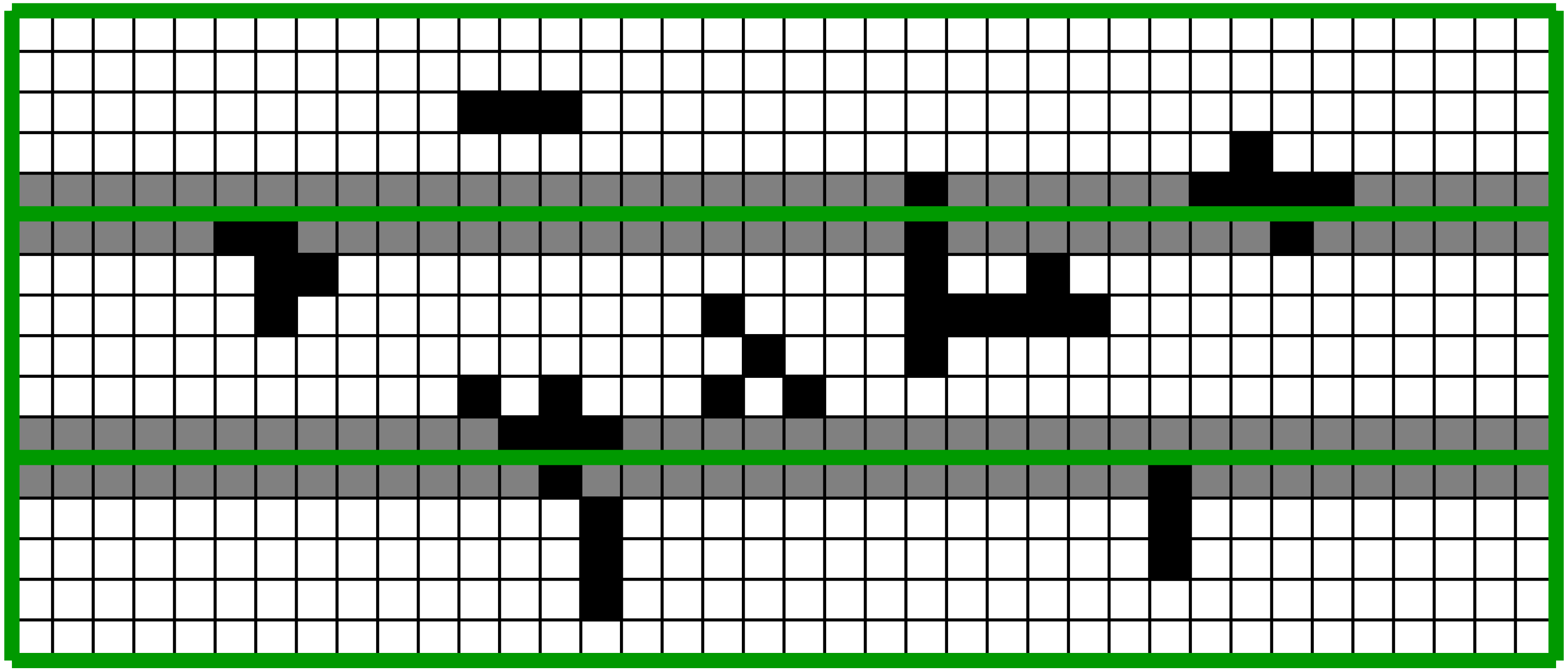
OpenMPI Implementation

- devide the world in t pieces, each for one process
- calculate cell by cell, generation by generation
- communication at the borders needed



OpenMPI Implementation

- devide the world in t pieces, each for one process
- calculate cell by cell, generation by generation
- communication at the borders needed
- synchronization after each step



Result Verification

The results of the OpenMP and MPI implementation need to be verifyed.

Result Verification

The results of the OpenMP and MPI implementation need to be verifyed.

- Output result state on command line (for small boards)

Result Verification

The results of the OpenMP and MPI implementation need to be verifyed.

- Output result state on command line (for small boards)
- comparison to (single-core) reference implementation (`golly`)

Result Verification

The results of the OpenMP and MPI implementation need to be verifyed.

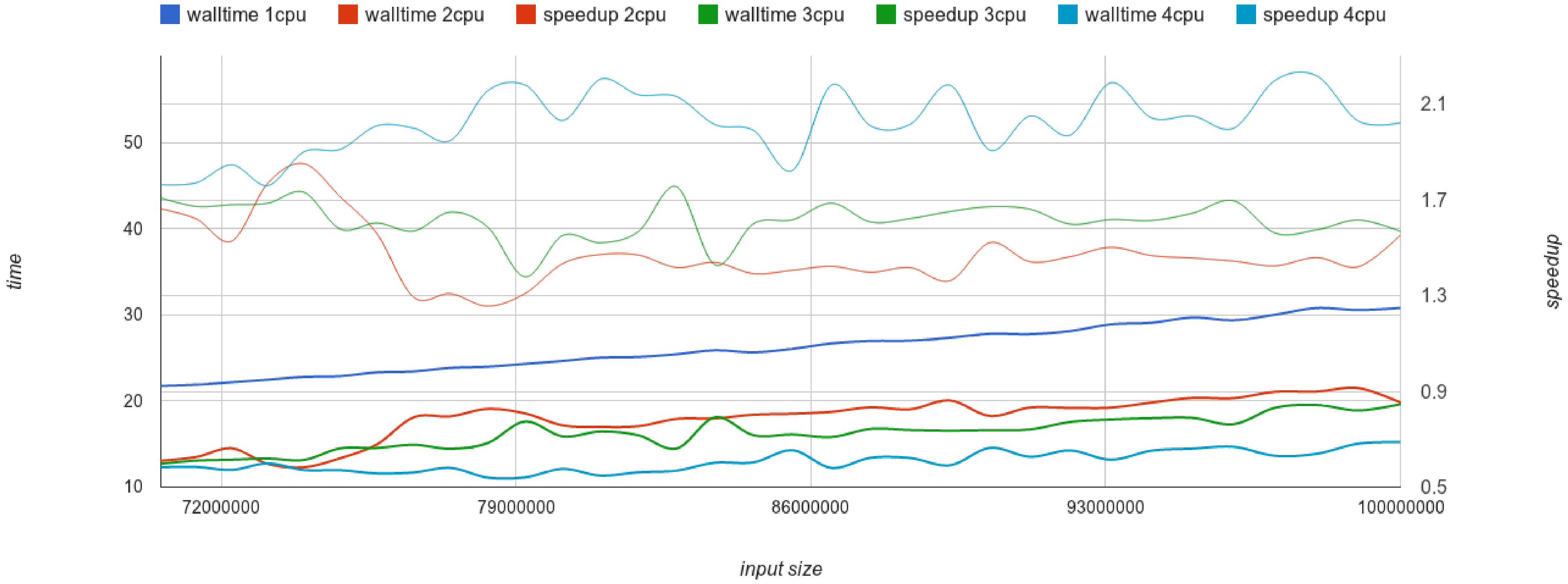
- Output result state on command line (for small boards)
- comparison to (single-core) reference implementation (`golly`)
- calculation of well-known patterns

Open MPI Input Size Benchmark

Compared to the OpenMP implementation, the Open MPI implementation shows a much worse speedup, even on a single machine. This is due to the message passing, which can be done through a network of nodes, but significantly slows down things.

Open MPI Input Size Benchmark

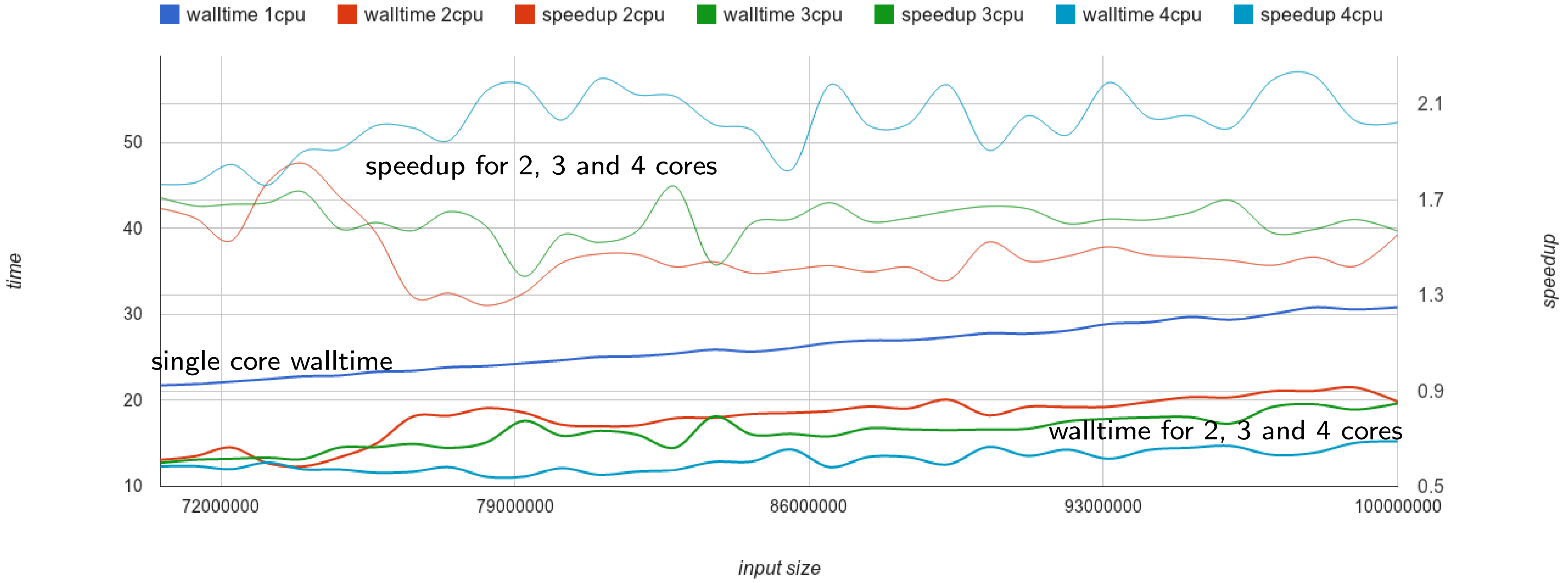
Compared to the OpenMP implementation, the Open MPI implementation shows a much worse speedup, even on a single machine. This is due to the message passing, which can be done through a network of nodes, but significantly slows down things.



Open MPI Input Size Benchmark

Compared to the OpenMP implementation, the Open MPI implementation shows a much worse speedup, even on a single machine. This is due to the message passing, which can be done through a network of nodes, but significantly slows down things.

test case: different board sizes, 4 generations of f-pentomino



speedup for 2, 3 and 4 cores

single core walltime

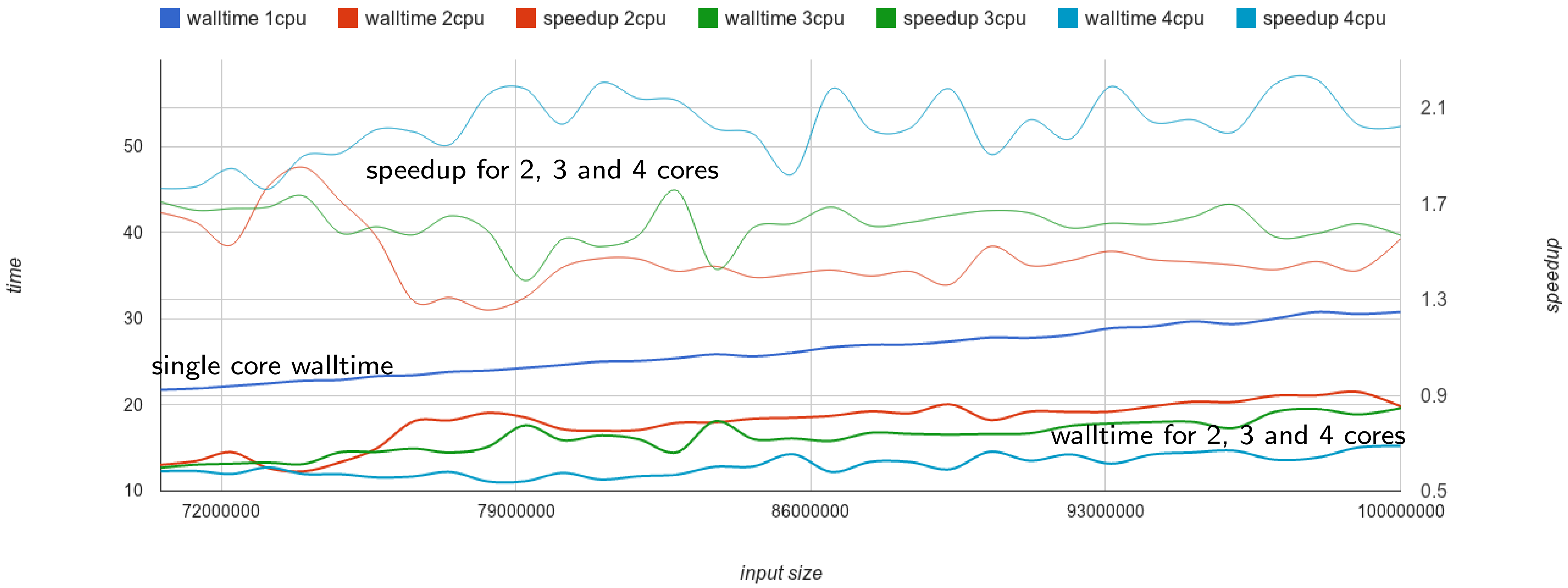
walltime for 2, 3 and 4 cores

Open MPI Input Size B

Compared to the OpenMP impler
plementation shows a much wors
machine. This is due to the me
done through a network of nodes
things.

- runtime still linear

test case: different board sizes, 4 generations of f-pentomino



speedup for 2, 3 and 4 cores

single core walltime

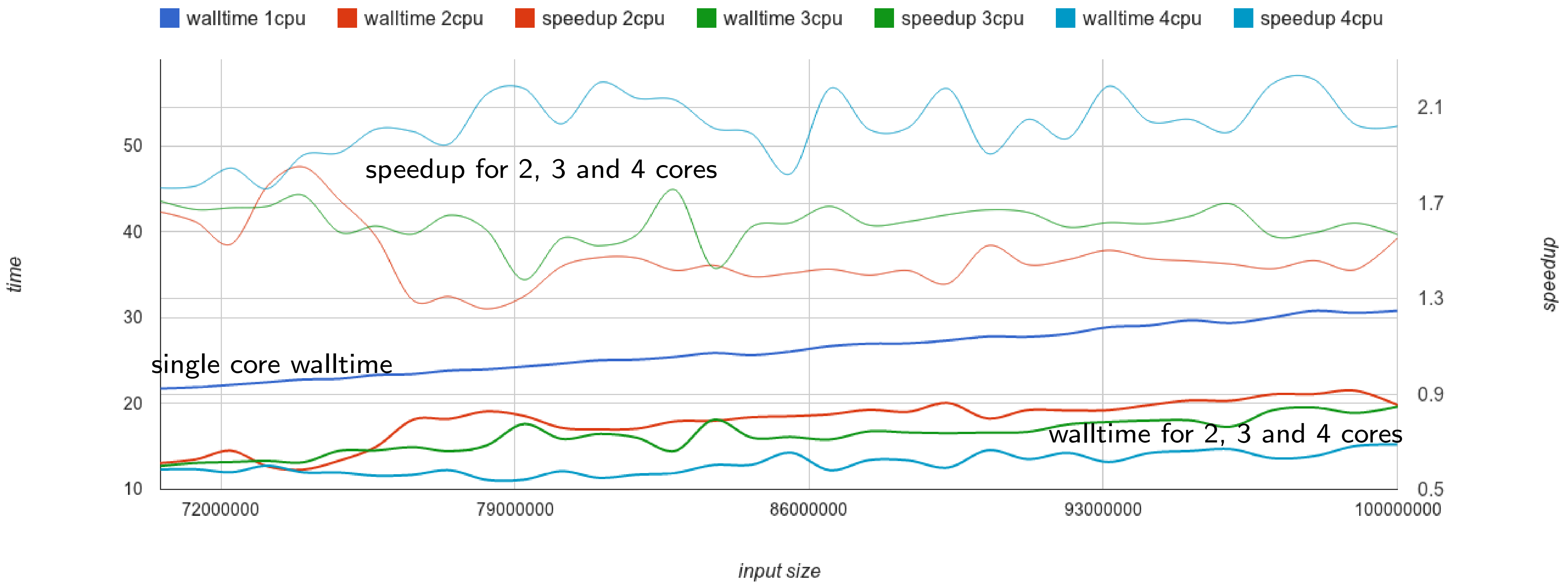
walltime for 2, 3 and 4 cores

Open MPI Input Size B

- runtime still linear
- speedup much worse than with OpenMP

Compared to the OpenMP impler
plementation shows a much wors
machine. This is due to the me
done through a network of nodes
things.

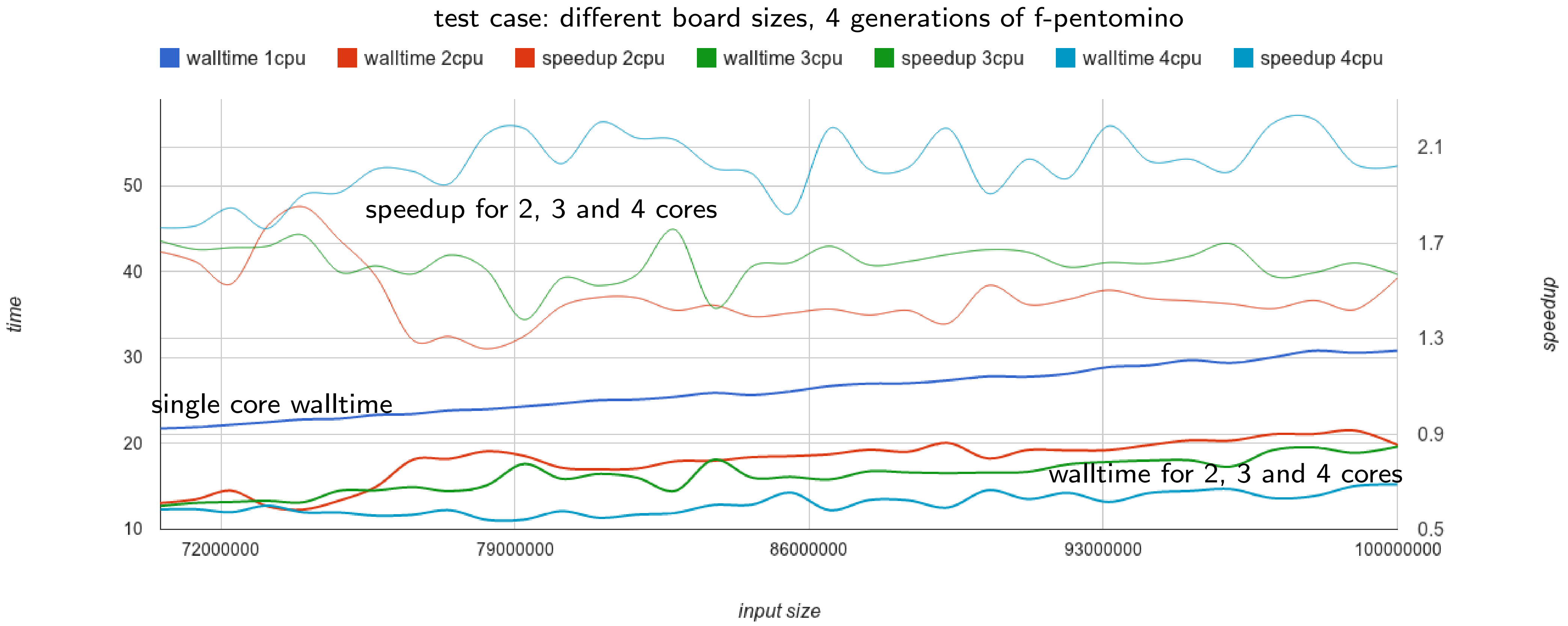
test case: different board sizes, 4 generations of f-pentomino



Open MPI Input Size B

Compared to the OpenMP impler
plementation shows a much wors
machine. This is due to the me
done through a network of nodes
things.

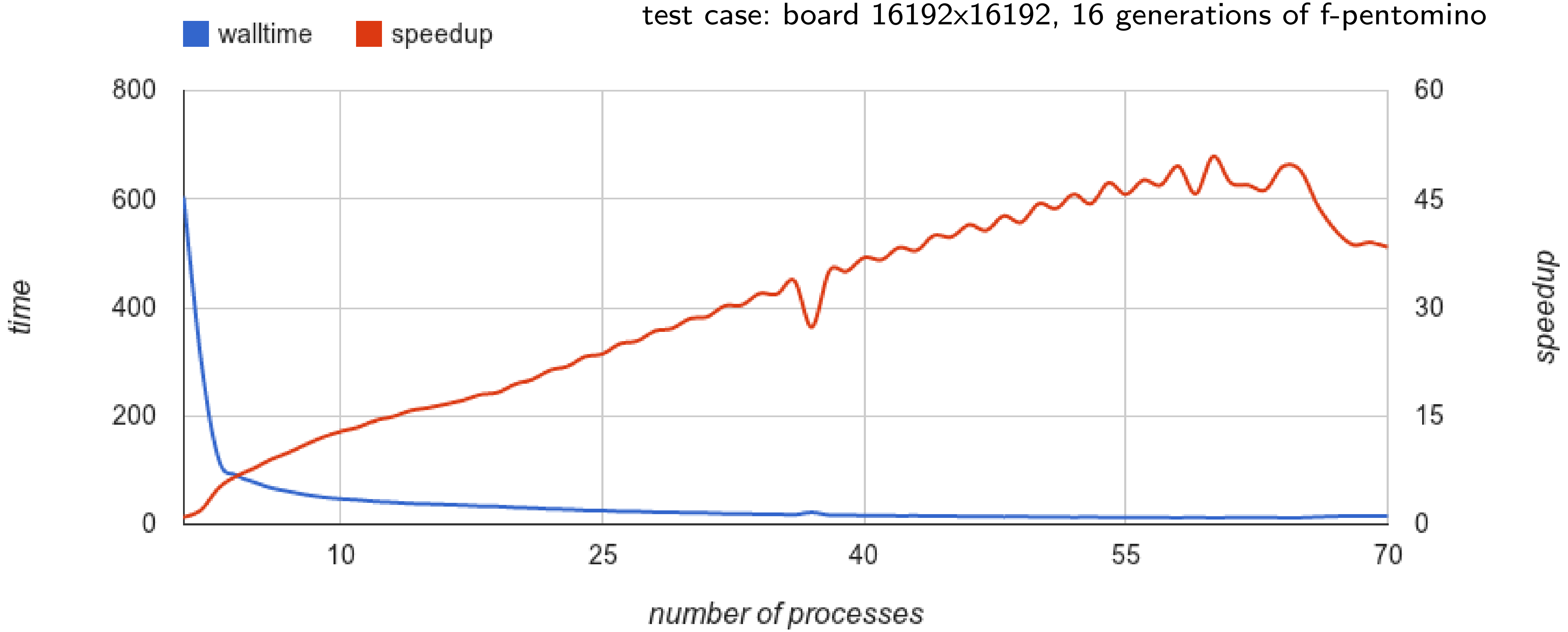
- runtime still linear
- speedup much worse than with OpenMP
- multi-process MPI runtime varies more than single-process runtime



Open MPI Process Number Benchmark

At CCR, the maximum speedup we can achive with a OpenMP implementation is 32. With the Open MPI implementation, we can achive greater speedups. For example, by using 32 2-core nodes, we can achieve up to 50 times the single-core speed.

test case: board 16192x16192, 16 generations of f-pentomino



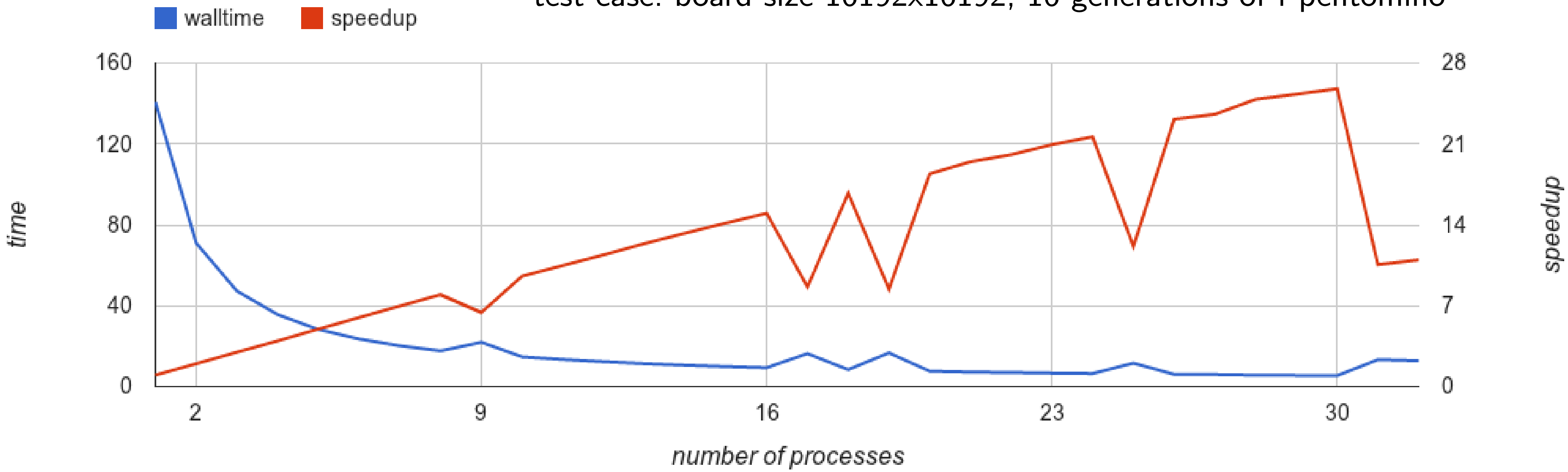
Open MPI Process Number Benchmark

Using machine with more cores, we can improve these results. Usings 4 8-core machines, we can achieve up to 26 times the single core computation speed:

Open MPI Process Number Benchmark

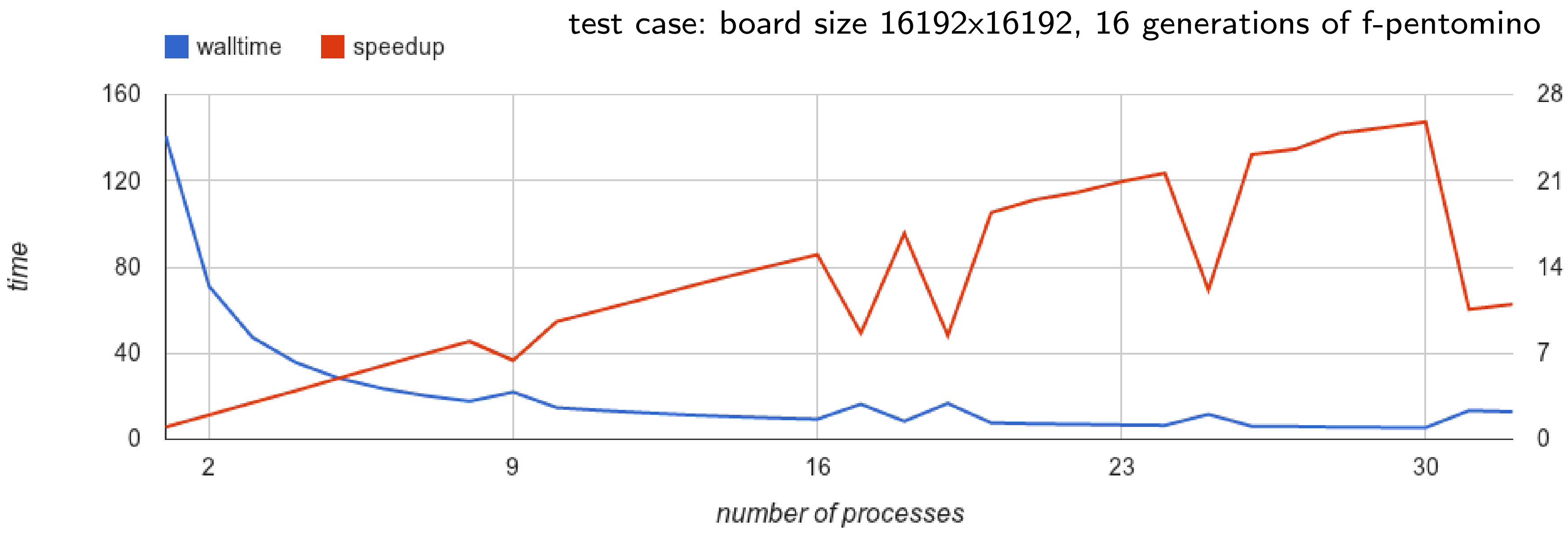
Using machine with more cores, we can improve these results. Usings 4 8-core machines, we can achieve up to 26 times the single core computation speed:

test case: board size 16192x16192, 16 generations of f-pentomino



Open MPI Process Number Benchmark

Using machine with more cores, we can improve these results. Usings 4 8-core machines, we can achieve up to 26 times the single core computation speed:



It's remarkable that for the first 8 tests, which all took place on a single machine with 8 cores, the speedup is almost optimal (that is, 7.96 when using 8 cores).

Further Improvements

Currently, the MPI implementation does not show any different runtimes for inputs of different kinds, but same size due to the basic implementation of Conway's rules.

Further Improvements

Currently, the MPI implementation does not show any different runtimes for inputs of different kinds, but same size due to the basic implementation of Conway's rules.

- improve runtime by not recalculating areas of the board, that did not get updated in the last generation

Further Improvements

Currently, the MPI implementation does not show any different runtimes for inputs of different kinds, but same size due to the basic implementation of Conway's rules.

- improve runtime by not recalculating areas of the board, that did not get updated in the last generation

Also, the current implementation does not consider how the nodes are connected.

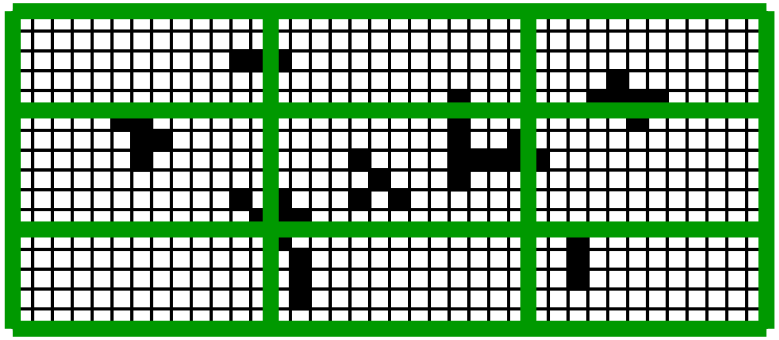
Further Improvements

Currently, the MPI implementation does not show any different runtimes for inputs of different kinds, but same size due to the basic implementation of Conway's rules.

- improve runtime by not recalculating areas of the board, that did not get updated in the last generation

Also, the current implementation does not consider how the nodes are connected.

- improve runtime by splitting the game's board into a grid which mirrors the structure of the cluster, in order to minimize waiting times



Conclusion and Future Work

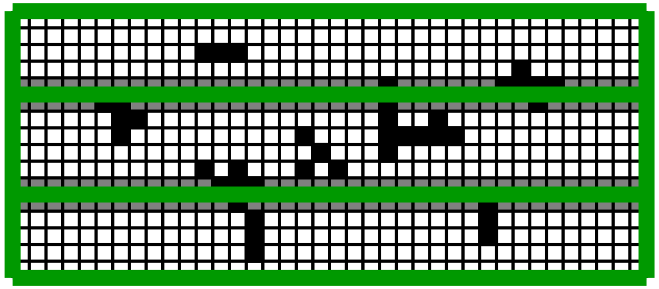
For neighbourhood count i,
cell $\leftarrow$ alive ? $i = 2$ or $i = 3$: $i = 3$;

- single core implementation is straightforward

Conclusion and Future Work

For neighbourhood count i,
cell ← alive ? $i = 2$ or $i = 3$: $i = 3$;

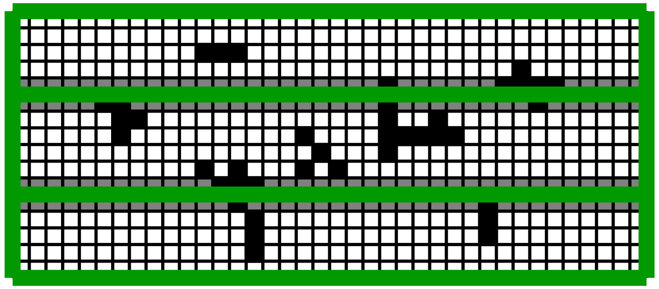
- single core implementation is straightforward
- OpenMP implementation achieves very good speedup, but is limited to machine size



Conclusion and Future Work

For neighbourhood count i,
cell $\longleftarrow$ alive ? $i = 2$ or $i = 3$: $i = 3$;

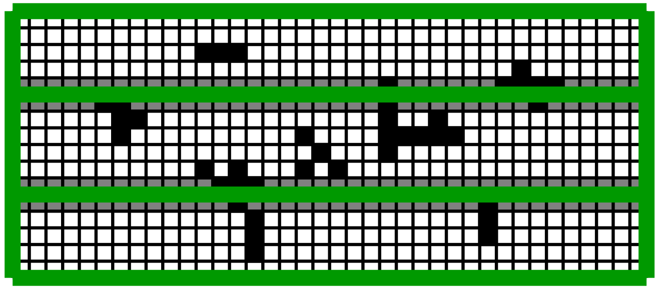
- single core implementation is straightforward
- OpenMP implementation achieves very good speedup, but is limited to machine size
- Open MPI is harder to implement, but can use more cores. In total, MPI achieves a better speedup, as we can use multiple nodes.



Conclusion and Future Work

For neighbourhood count i,
cell $\leftarrow$ alive ? $i = 2$ or $i = 3$: $i = 3$;

- single core implementation is straightforward
- OpenMP implementation achieves very good speedup, but is limited to machine size
- Open MPI is harder to implement, but can use more cores. In total, MPI achieves a better speedup, as we can use multiple nodes.

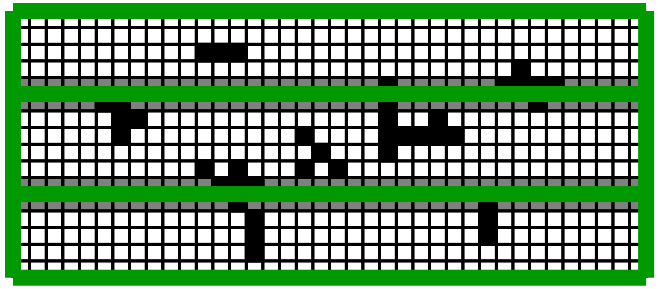


- For future work, cuda could be useful to process as many cells as possible in parallel

Conclusion and Future Work

For neighbourhood count i,
cell ← alive ? $i = 2$ or $i = 3$: $i = 3$;

- single core implementation is straightforward
- OpenMP implementation achieves very good speedup, but is limited to machine size
- Open MPI is harder to implement, but can use more cores. In total, MPI achieves a better speedup, as we can use multiple nodes.

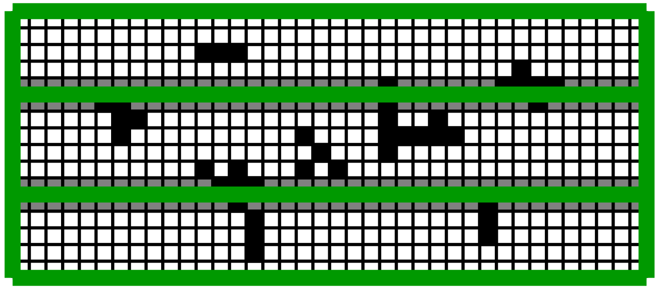


- For future work, cuda could be useful to process as many cells as possible in parallel
- Also, improving the MPI implementation by considering the grid structure will give better speedup

Conclusion and Future Work

For neighbourhood count i,
cell $\leftarrow$ alive ? $i = 2$ or $i = 3$: $i = 3$;

- single core implementation is straightforward
- OpenMP implementation achieves very good speedup, but is limited to machine size
- Open MPI is harder to implement, but can use more cores. In total, MPI achieves a better speedup, as we can use multiple nodes.



- For future work, cuda could be useful to process as many cells as possible in parallel
- Also, improving the MPI implementation by considering the grid structure will give better speedup
- Engine should be extended in a way that can simulate other cellular automatons