

Parallel Computing for Science & Engineering SSC 374/394c

Spring 2015

Instructors:

Victor Eijkhout, Research Scientist, TACC

Cyrus Proctor, Research Associate, TACC



THE UNIVERSITY OF TEXAS AT AUSTIN
Texas Advanced Computing Center

The ideas of parallel programming

- As illustrated by Conway's *Game of Life*
- <http://youtu.be/C2vgICfQawE>
- This has the same structure as certain important applications (e.g., PDEs) but requires no math to explain.
- Note: this is about parallel *programming*, not so much about parallel *hardware*

How do you code this?

- First the function for updating a single cell

```
def life_evaluation( cells ):
    # cells is a 3x3 array
    count = 0
    for i in [0,1,2]:
        for j in [0,1,2]:
            if i!=1 and j!=1:
                count += cells[i,j]
    cells[1,1] = life_count_evaluation(count )

def life_count_evaluation( count ):
    # big if statement
    return newval
```

How do you code this?

- Now to update the whole board
- One time-stepping loop
- Two loops for the board

```
life_board.create(final_time,N,N)

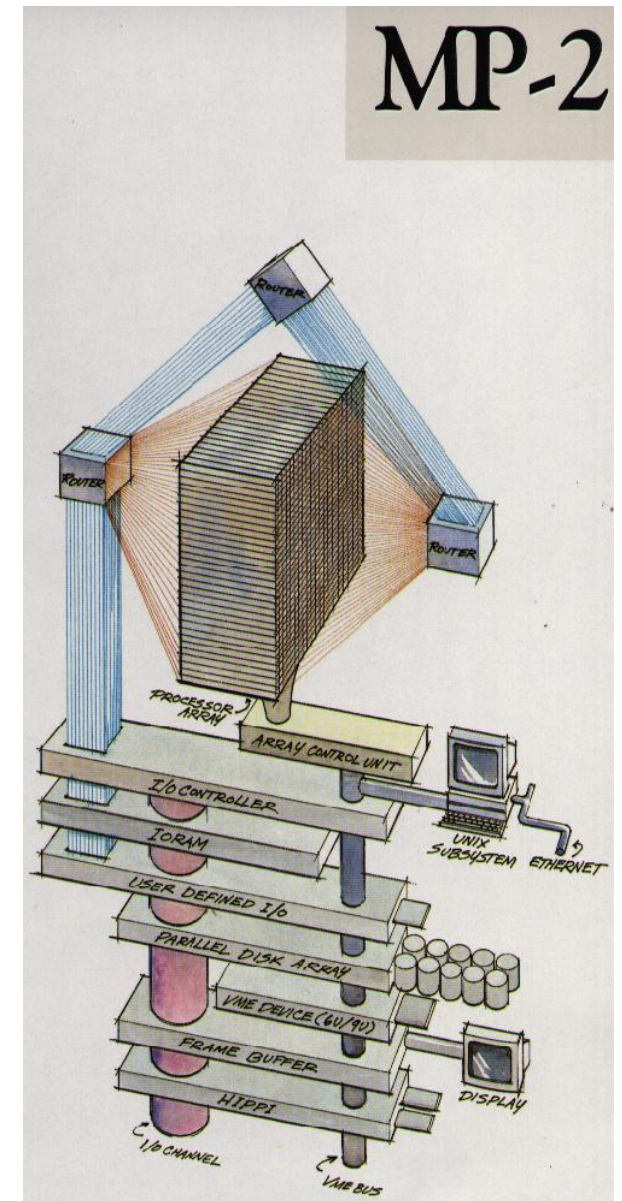
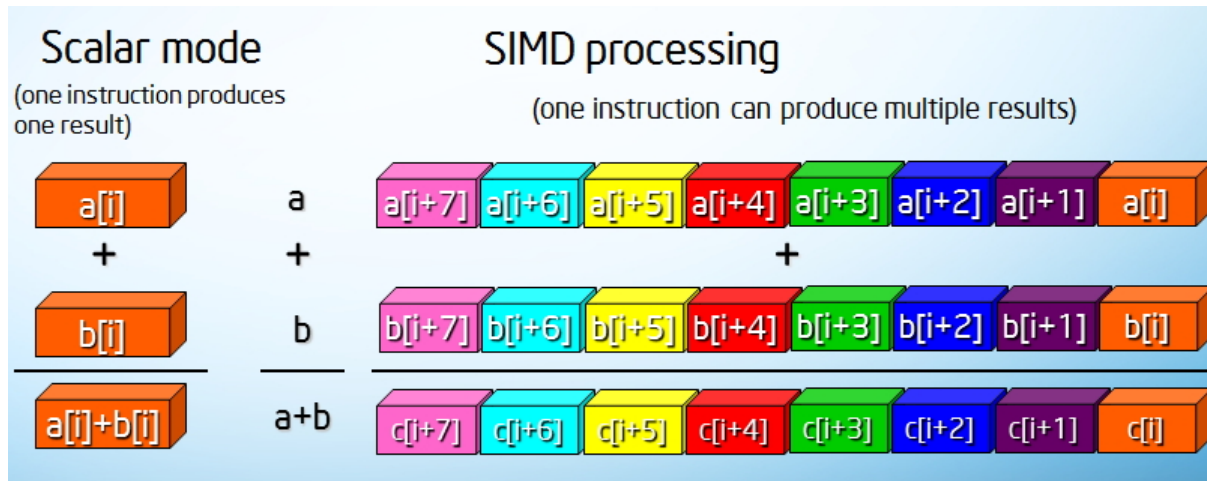
for t in [0:final_time-1]:
    for i in [0:N-1]:
        for j in [0:N-1]:
            life_board[t+1,i,j] :=
                life_evaluation(life_board
                               [t,i-1:i+1,j-1:j+1] )
```

Where does parallelism come from?

- The program text specifies a sequence of operations
- However, some operations can be done in any order
- => so they can also be done simultaneously
- There are no compilers that recognize this, so you have to code it by hand and that's what you will learn here....

Data parallelism

- Independent data items, each undergoing the same operation
- Then: “array processors”
- Now: vector instructions



SIMD parallelism

- Single Instruction Multiple Data
- Simplified instruction handling: only one instruction fetch/decode/whatever for multiple data items
- Need to have many independent operations (examples?)
- Data storage may need to be regular

Example: GPUs

- Graphical Processing Units are SIMD-like (not completely lockstep)
- Programmed in CUDA:
kernel contains sequential code,
kernel is executed in parallel

```
kerneldef life_step( board ):  
    i = my_i_number()  
    j = my_j_number()  
    board[i,j] = life_evaluation  
                  ( board[i-1:i+1,j-1:j+1] )  
  
for t in [0:final_time]:  
    <<N,N>>life_step( board )
```


Parallel programming may mess up your code!

- Parallelism on the instruction level:
innermost loop
- Sometimes loop exchange needed

```
for i=1,N:
  for j=1,N:
    count = 0
    for h in [-1,0,1]:
      for v in [-1,0,1]:
        count += cell[i+v,j+h]
```

```
for i=1,N:
  for j=1,N:
    count[i,j] = 0
  for h in [-1,0,1]:
    for v in [-1,0,1]:
      for i=1,N:
        for j=1,N:
          count += cell[i+v,j+h]
```

Minimal intervention: loop parallelism

- Loops are an important source of parallelism
- Parallelize by indicating what loops parallel

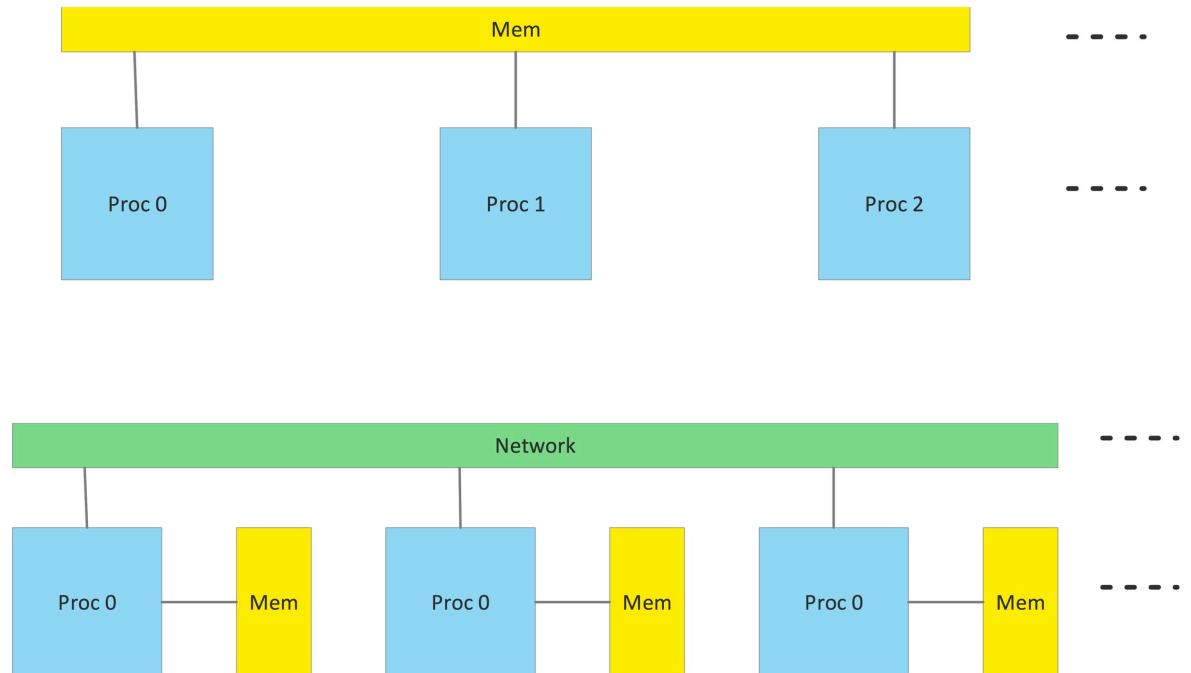
```
def life_generation( board,tmp ):  
    # OMP parallel for  
    for i in [0:N-1]:  
        for j in [0:N-1]:  
            tmp[i,j] = board[i,j]  
    # OMP parallel for  
    for i in [0:N-1]:  
        for j in [0:N-1]:  
            board[i,j] = life_evaluation  
                ( tmp[i-1:i+1,j-1:j+1] )
```

Granularity of parallelism

- So far: independence of single operations / single data points:
fine-grained parallelism
- Locality: points close together should be handled by the same processor
- Process the board by lines or subparts:
coarse-grained parallelism

Why coarse-grained parallelism?

- Shared memory: every processor can find every data item
- Distributed memory: some data is local, other not
- Locality



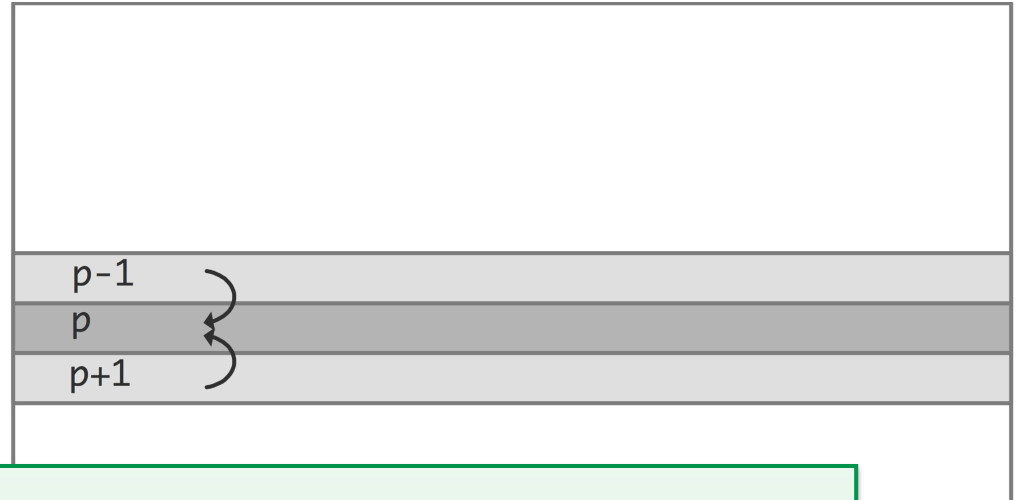
What does distributed memory look like?

- Stampede
- 160 cabinets,
6400 nodes,
500k cores....



How do you program distributed memory?

- Explicit message passing



```
p = my_processor_number()

high_line = MPI_Receive(from=p-1,cells=N)
low_line = MPI_Receive(from=p+1,cells=N)

tmp_line = my_line.copy()
my_line = life_line_update(high_line,tmp_line,low_line,N)
```

No, really.....

- You can't receive without someone else sending
- But the someone else is running the same program...
- Single Program Multiple Data:
each processor runs the same program,
just on different data:
- Execution differs in:
loop bounds,
branches of conditionals

Two-sided message passing

- Everyone does both send and receive calls
- Attempt at coding this:

- And even this is not correct

```
p = my_processor_number()

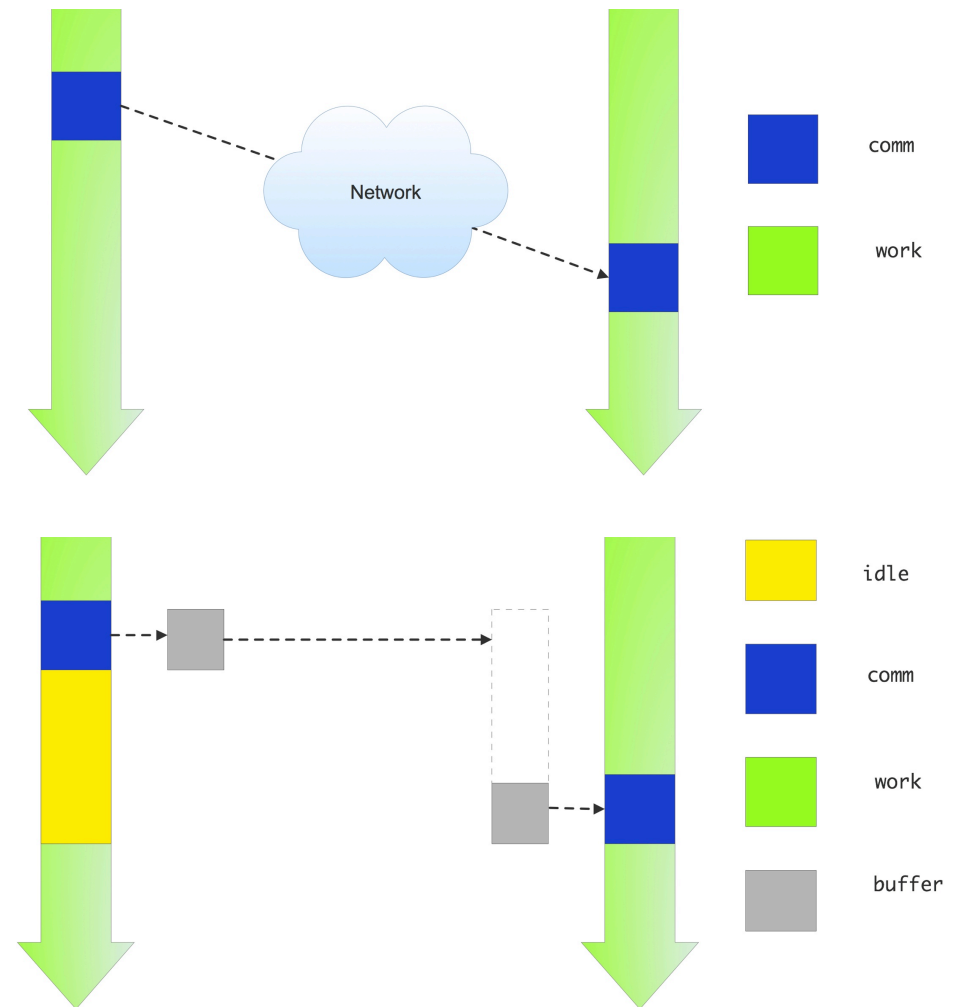
# send my data
my_line.MPI_Send(to=p-1,cells=N)
my_line.MPI_Send(to=p+1,cells=N)

# get data from neighbours
high_line = MPI_Receive(from=p-1,cells=N)
low_line = MPI_Receive(from=p+1,cells=N)
tmp_line = my_line.copy()

# do the local computation
my_line = life_line_update
                (high_line,tmp_line,low_line,N)
```


Blocking communication

- Data has to be somewhere
- You can only send if someone else receives
- Deadlock possible if everyone is receiving, no one is sending

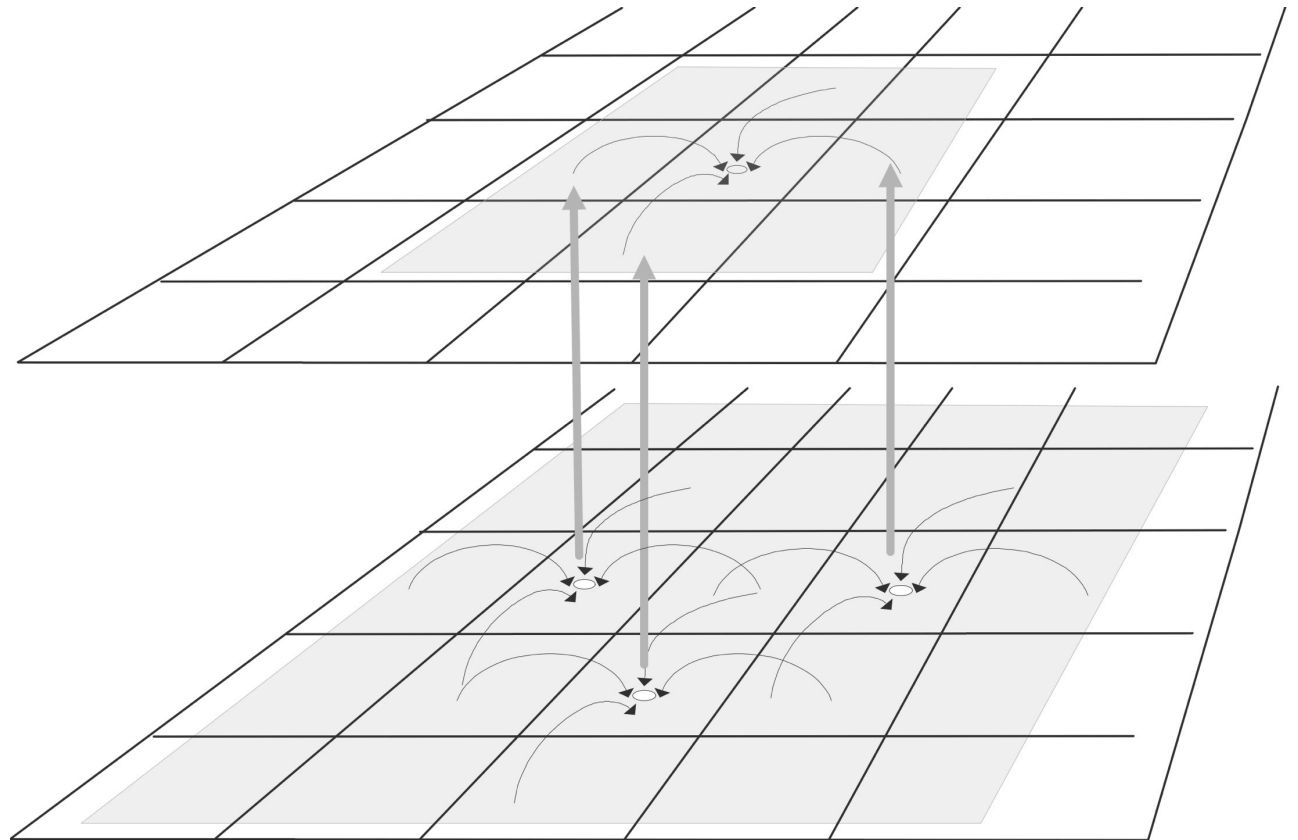


Task parallelism

- Think about instructions rather than data
- In the Game of Life there are N^2T updates
- How independent are they?

Life updates dependencies

- Cell needs a box around it
- Each cell in the box needs....
- => Cone of influence



Task scheduling

- User indicates dependencies
- Algorithm under the hood matches available tasks to available processors/cores

```
while there_are_tasks_left():  
    for r in running_tasks:  
        if r.finished():  
            for t in scheduled_tasks:  
                t.mark_input_available(r)  
            t = find_available_task()  
            p = find_available_processor()  
            schedule(t,p)
```

```
for t in [0:T]:  
    for i in [0:N]:  
        for j in [0:N]:  
            task( id=[t+1,i,j],  
                  prereqs=[  
                      [t,i,j],  
                      [t,i-1,j],  
                      [t,i+1,j]  
                      # et cetera  
                  ] )
```

Summary

- SIMD / vector parallelism:
very fine-grained
vector instructions
- Loop-based parallelism:
OpenMP directives
- Tasks:
OpenMP tasks, medium grain
- Message passing:
MPI, coarse grain