Why Structured Parallel Programming Matters

Murray Cole

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

Edinburgh

Edinburgh

Scotland's sunshine capital!

What is Unstructured Parallel Programming?

What is Unstructured Parallel Programming?

Simple parallel programming frameworks (Posix threads, core MPI) are universal.

What is Unstructured Parallel Programming?

Simple parallel programming frameworks (Posix threads, core MPI) are universal.

They can be used to describe arbitrarily complex and dynamically determined interactions between activities.

What is Unstructured Parallel Programming?

Simple parallel programming frameworks (Posix threads, core MPI) are universal.

They can be used to describe arbitrarily complex and dynamically determined interactions between activities.

Programming is by careful selection and combination of operations drawn from a small, simple set.

What is Unstructured Parallel Programming?

It is difficult for programmers, examining such a program statically, to understand the overall pattern involved (if such exists), and to radically revise it.

What is Unstructured Parallel Programming?

It is difficult for programmers, examining such a program statically, to understand the overall pattern involved (if such exists), and to radically revise it.

It is very difficult for implementation mechanisms (working statically and/or dynamically) to attempt optimisations which work beyond single instances of the primitives.

Patterns in Parallel Computing

Patterns in Parallel Computing

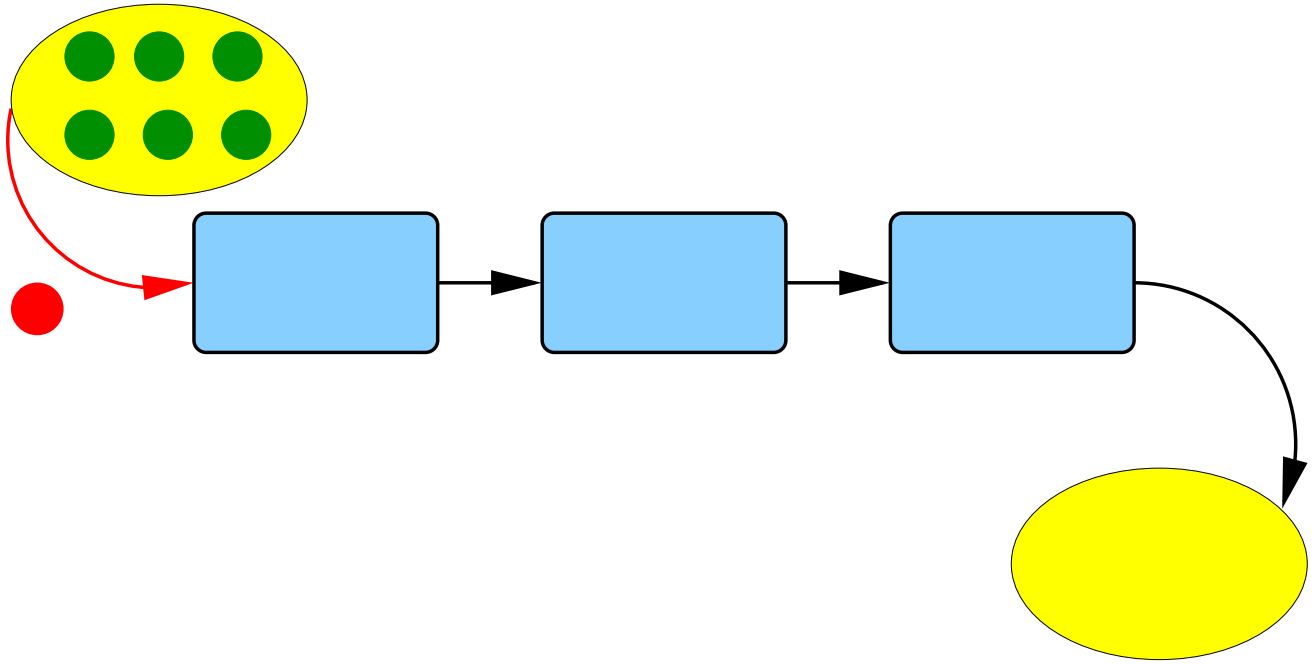
Many (most?) parallel applications don't actually involve arbitrary, dynamic interaction patterns.

Patterns in Parallel Computing

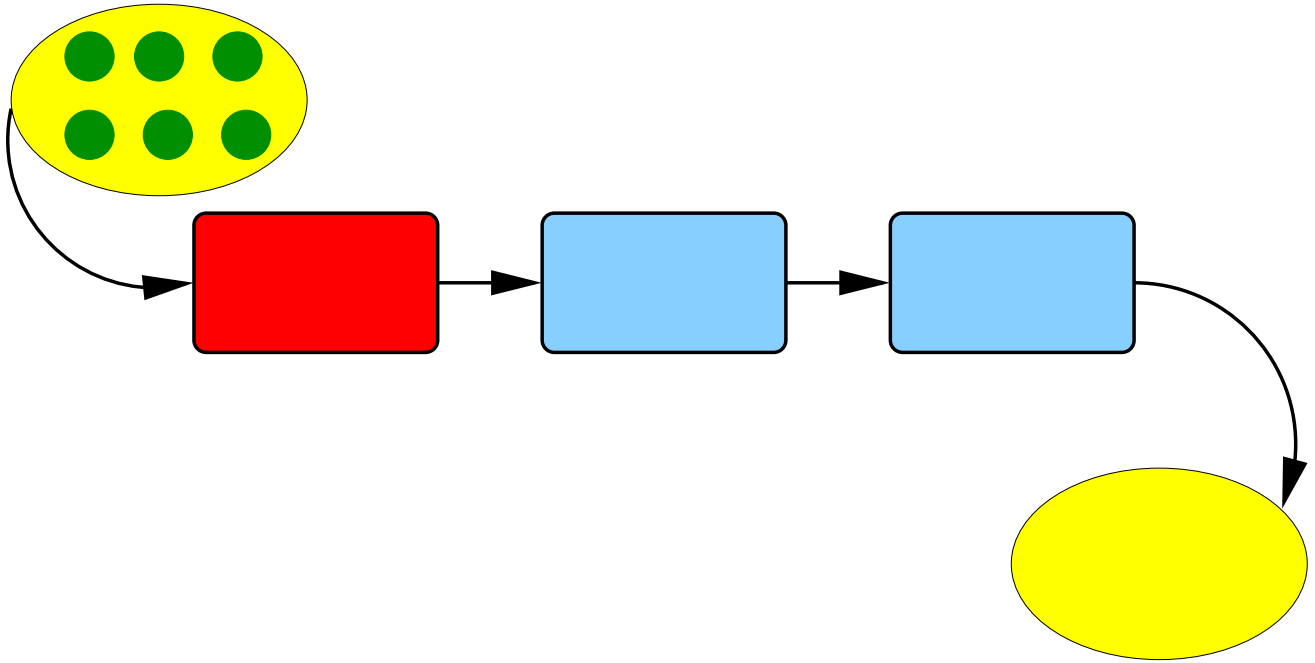
Many (most?) parallel applications don't actually involve arbitrary, dynamic interaction patterns.

Sometimes the pattern is entirely pre-determined.

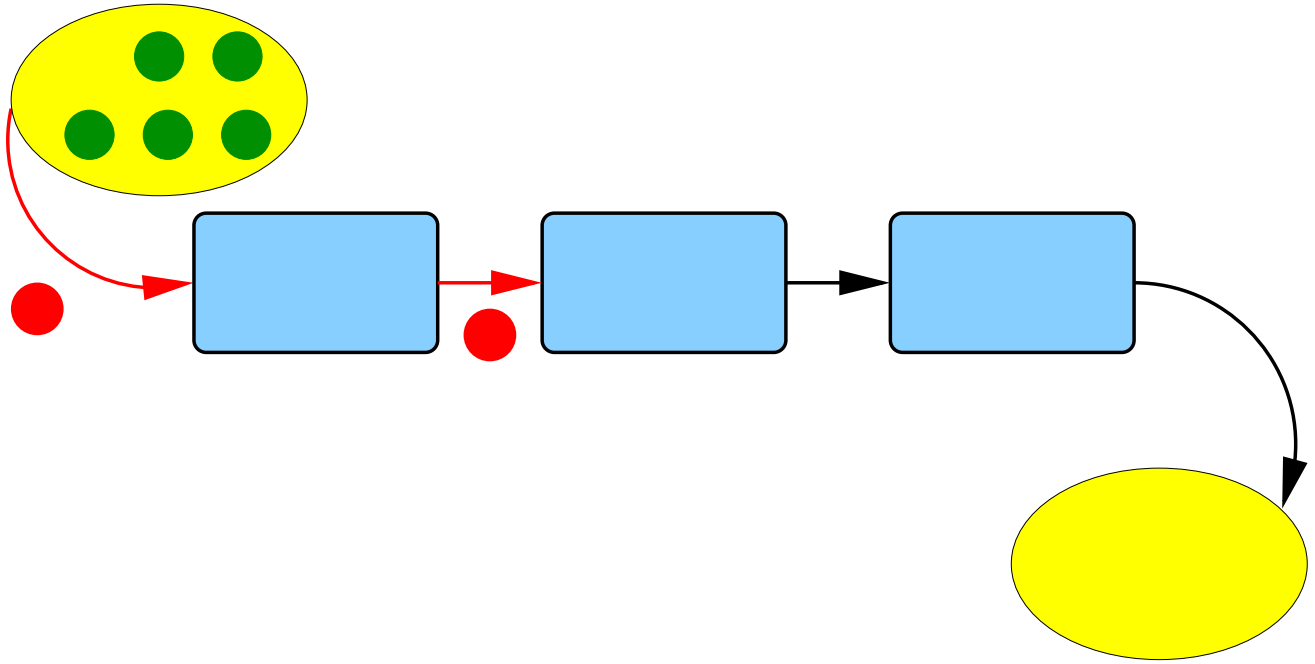
A Pipeline



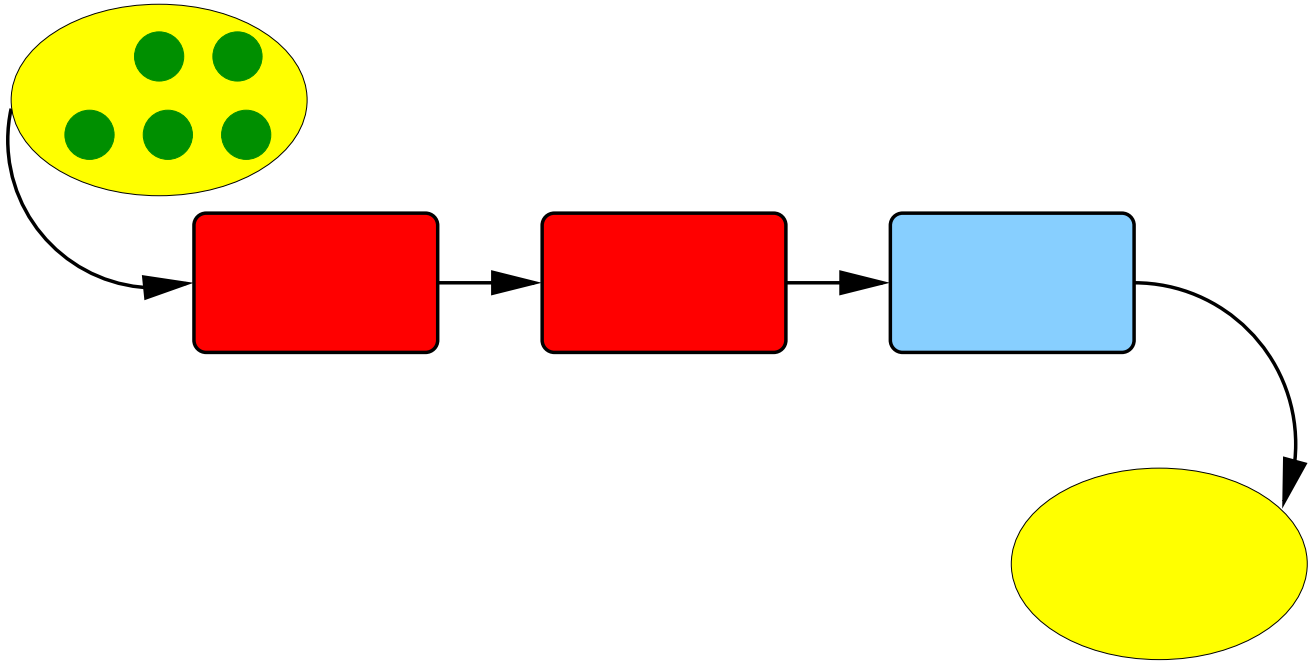
A Pipeline



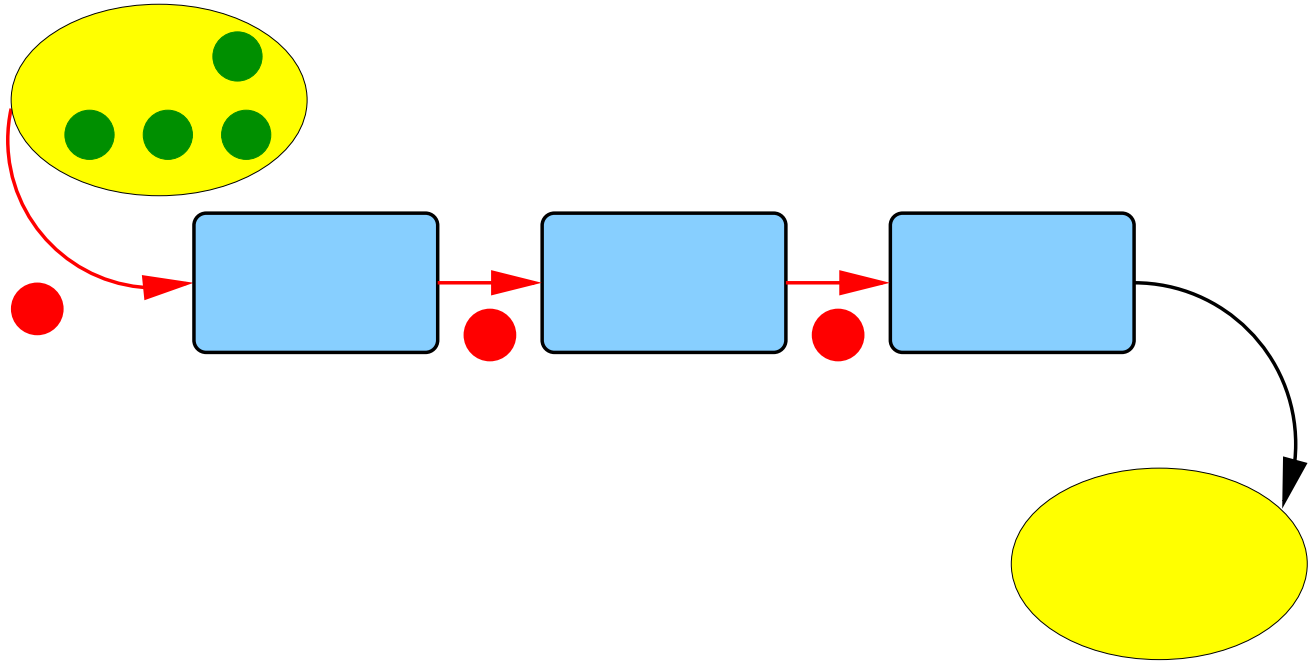
A Pipeline



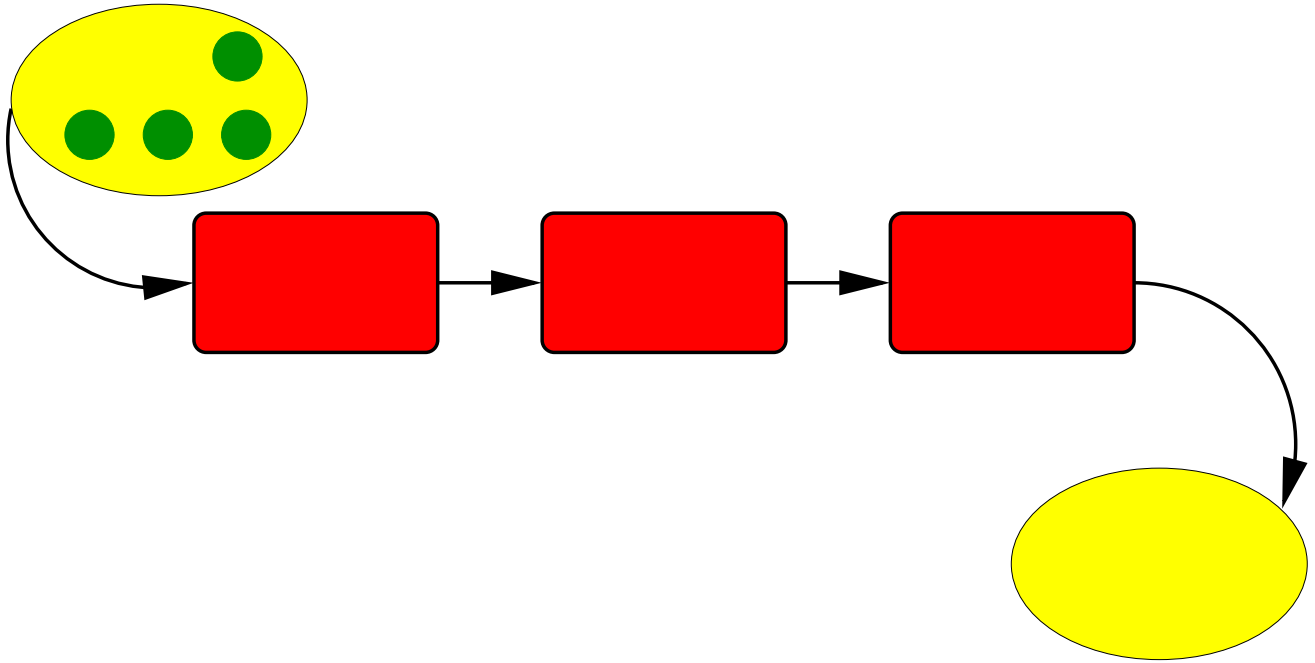
A Pipeline



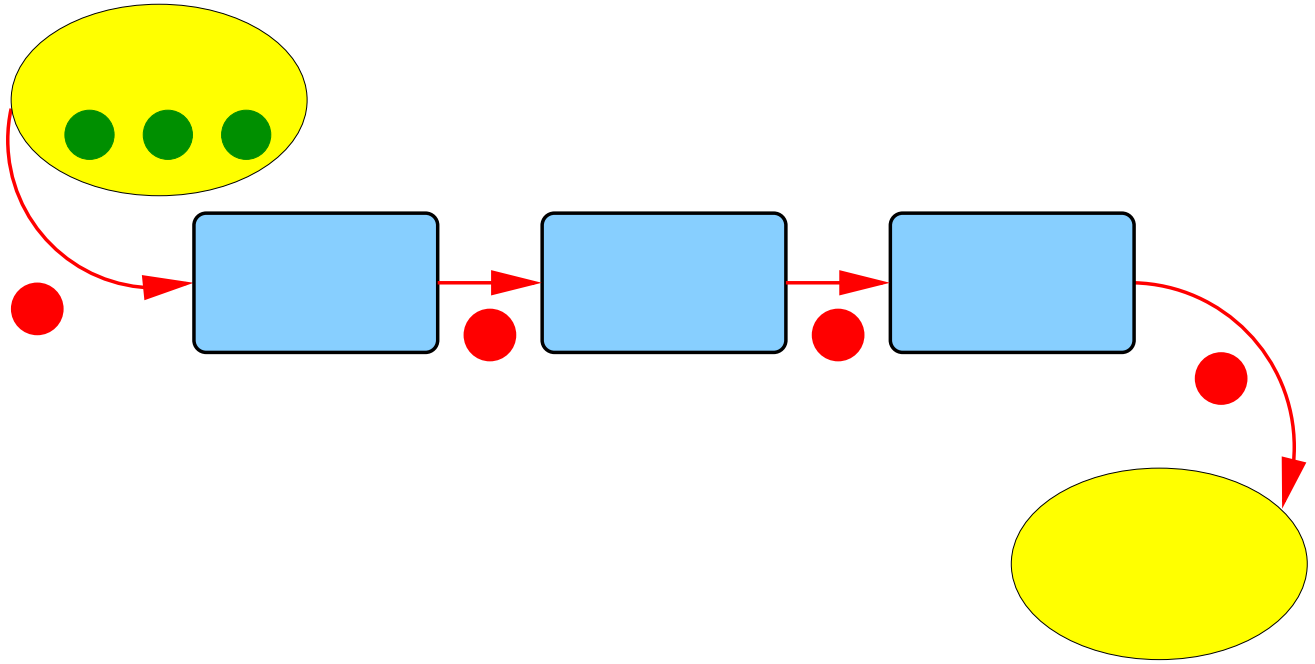
A Pipeline



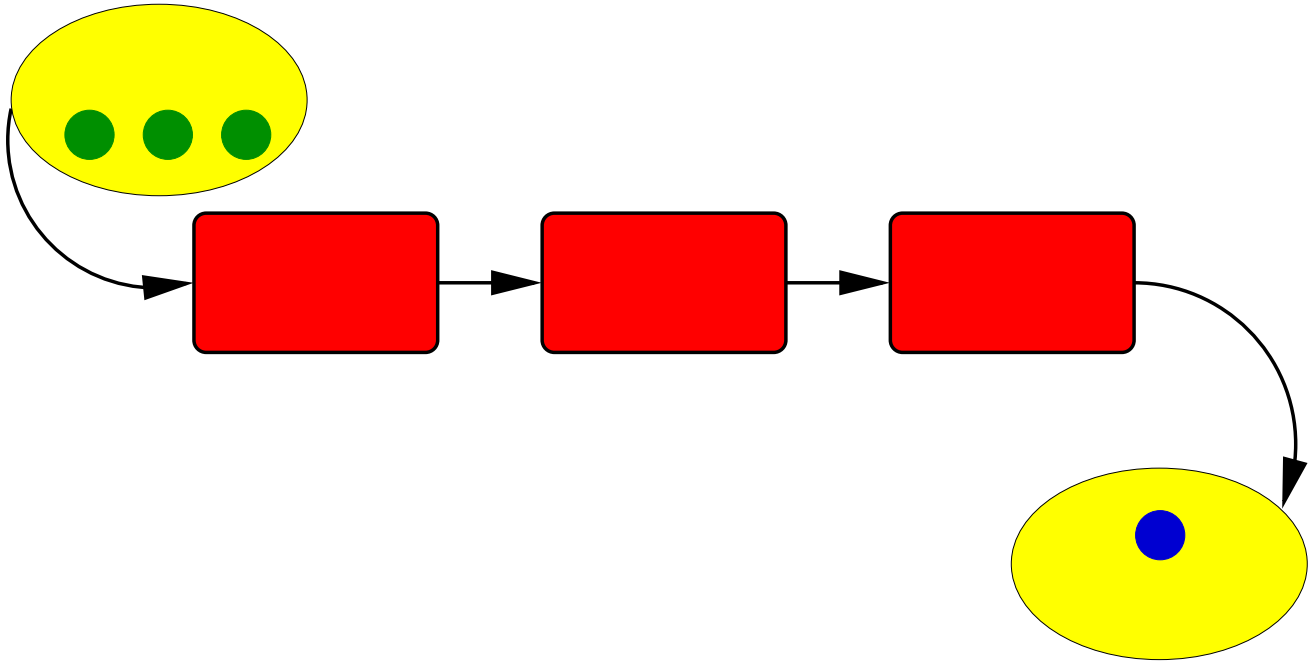
A Pipeline



A Pipeline



A Pipeline



Patterns in Parallel Computing

Many (most?) parallel applications don't actually involve arbitrary, dynamic interaction patterns.

Sometimes the pattern is entirely pre-determined.

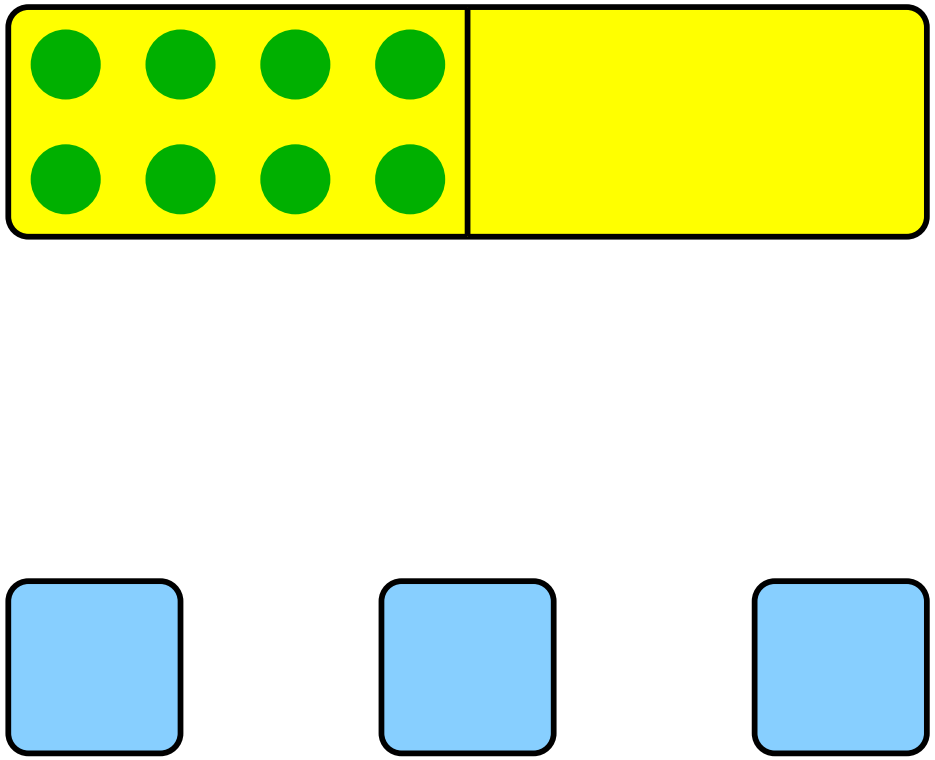
Patterns in Parallel Computing

Many (most?) parallel applications don't actually involve arbitrary, dynamic interaction patterns.

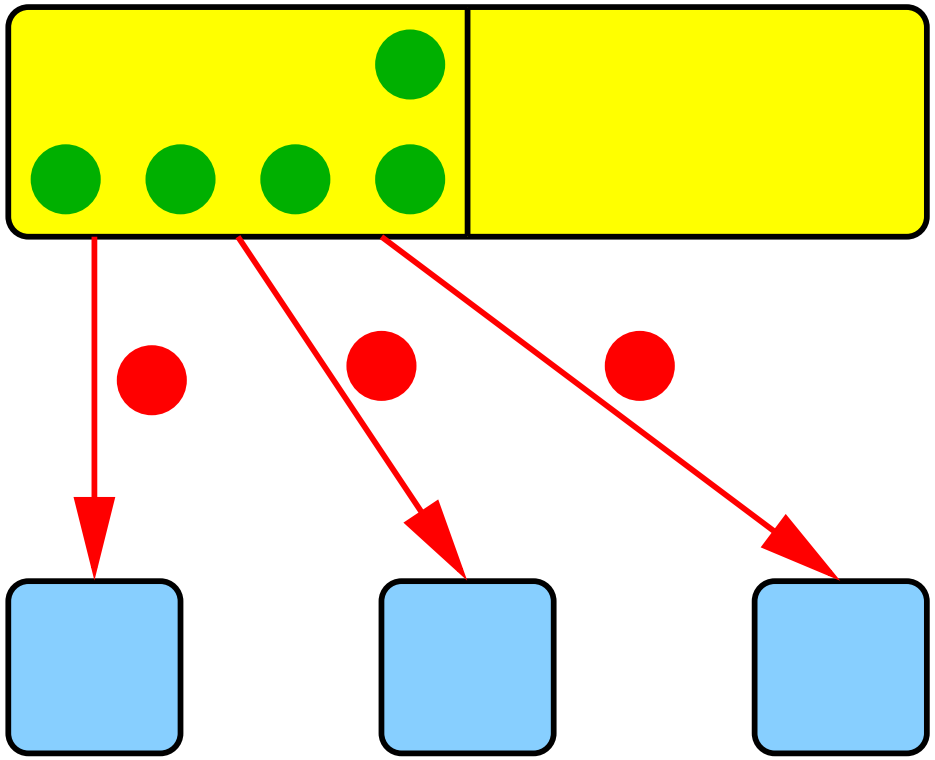
Sometimes the pattern is entirely pre-determined.

Sometimes non-determinism is constrained within a wider pattern.

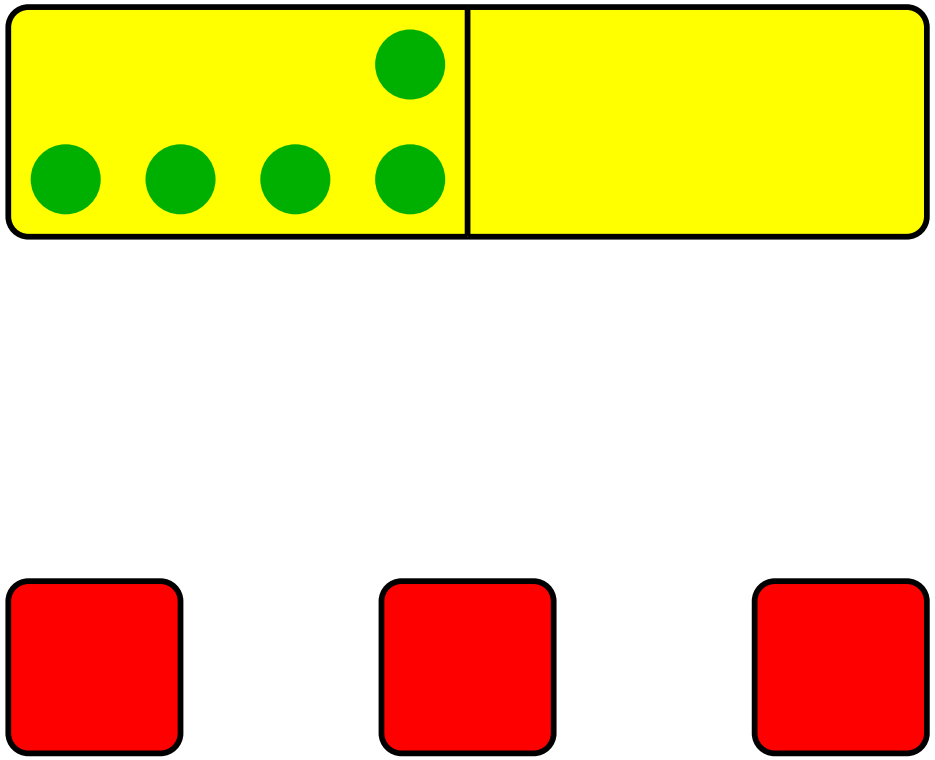
A Task Farm



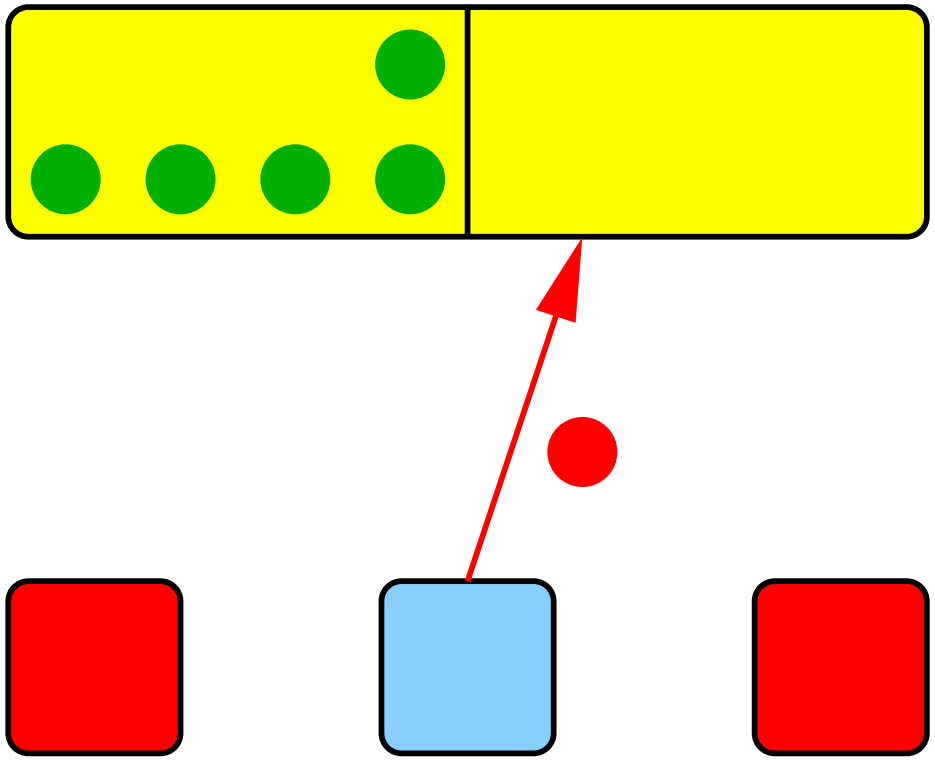
A Task Farm



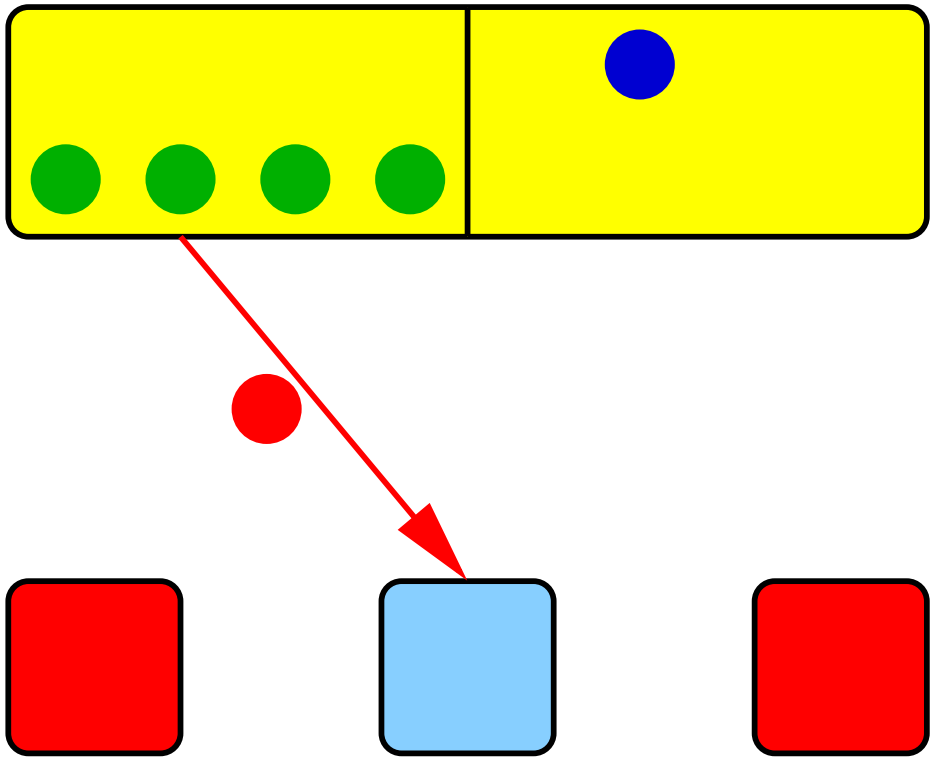
A Task Farm



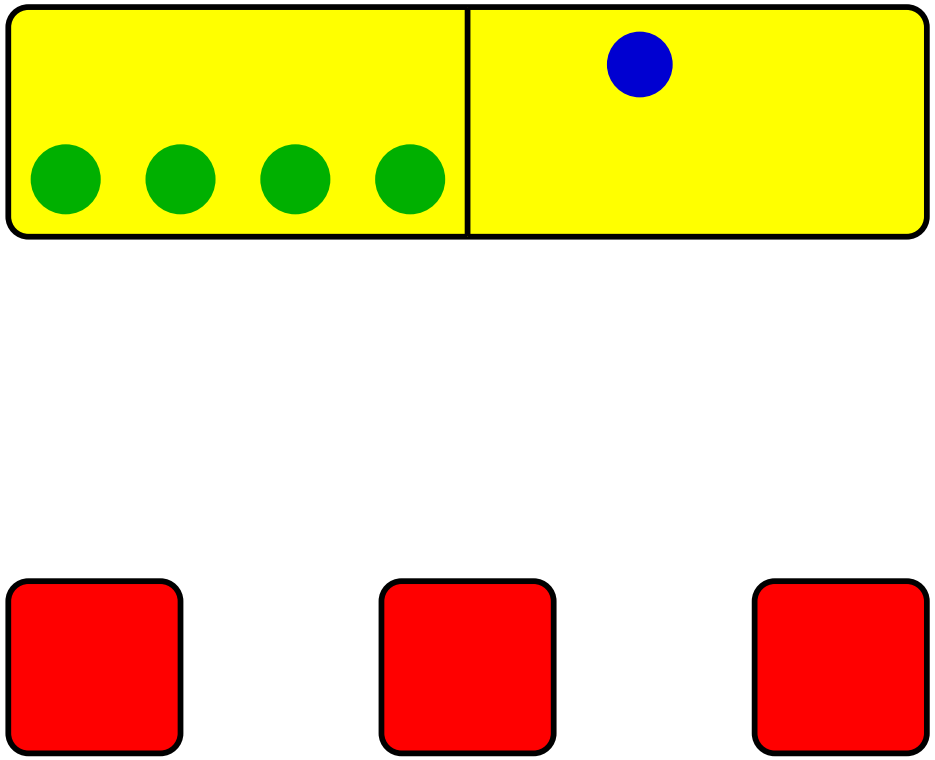
A Task Farm



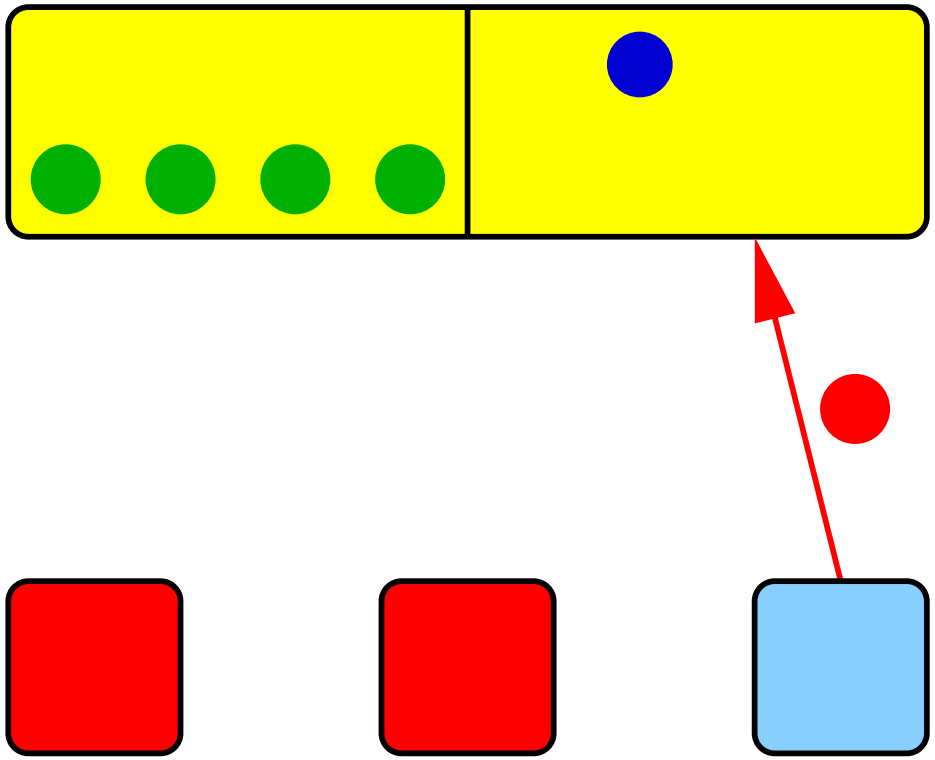
A Task Farm



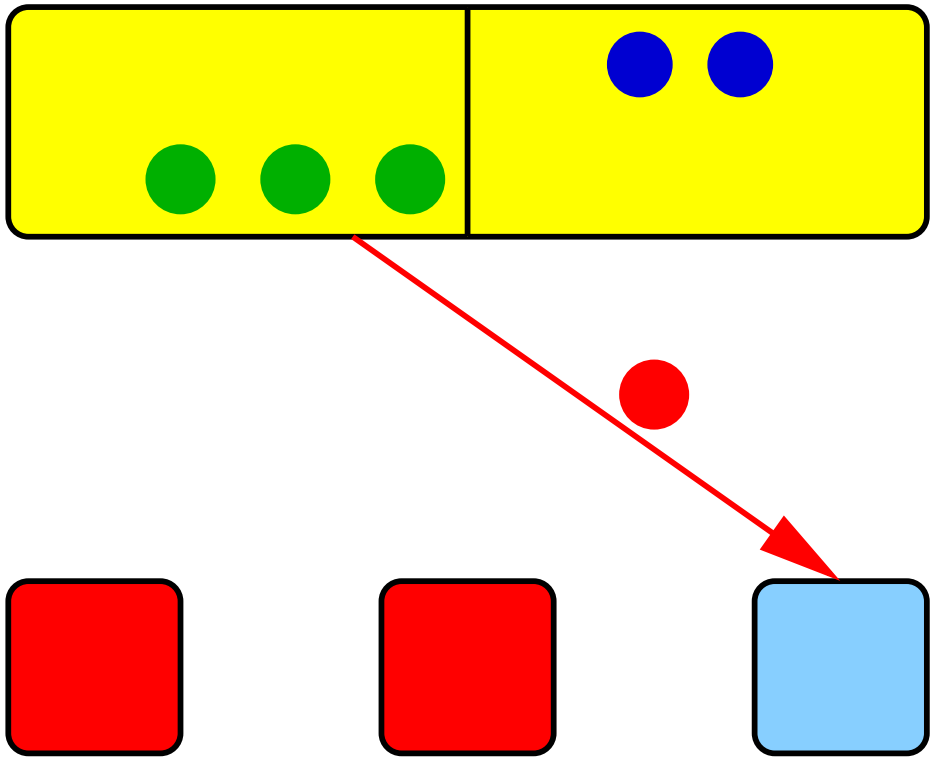
A Task Farm



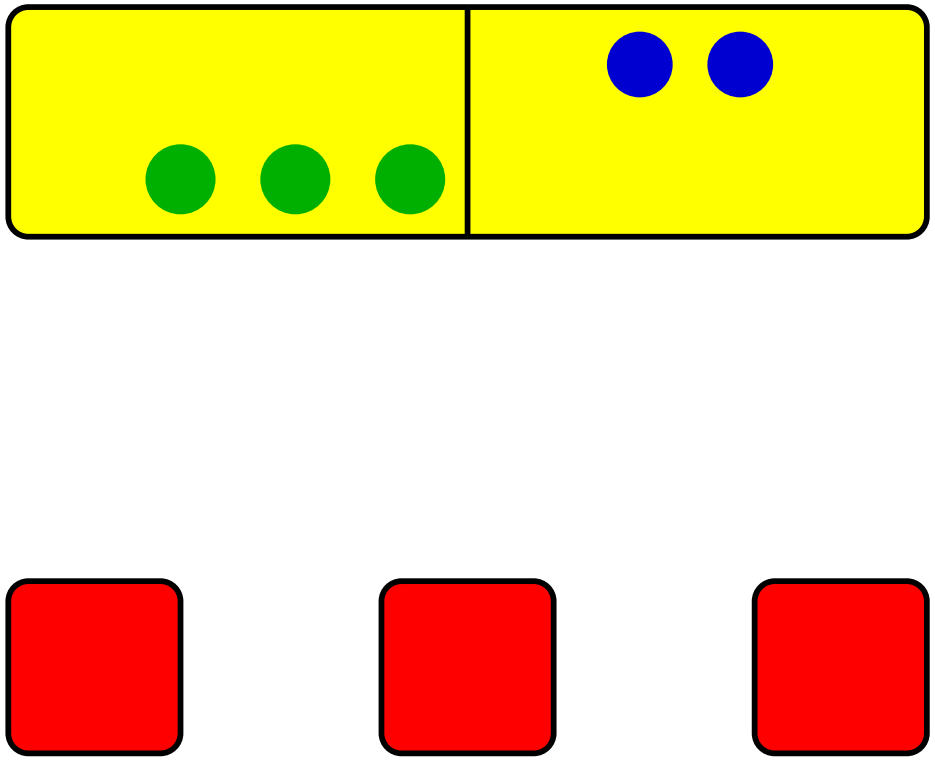
A Task Farm



A Task Farm



A Task Farm



Patterns in Parallel Computing

Many (most?) parallel applications don't actually involve arbitrary, dynamic interaction patterns.

Sometimes the pattern is entirely pre-determined.

Sometimes non-determinism is constrained within a wider pattern.

Patterns in Parallel Computing

Many (most?) parallel applications don't actually involve arbitrary, dynamic interaction patterns.

Sometimes the pattern is entirely pre-determined.

Sometimes non-determinism is constrained within a wider pattern.

The use of an unstructured parallel programming mechanism prevents the programmer from expressing information about the pattern - it remains implicit in the collected uses of the simple primitives.

What is Structured Parallel Programming?

What is Structured Parallel Programming?

The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library functions, control constructs or similar, so that application programmers can explicitly declare that the application follows one or more such patterns.

What is Structured Parallel Programming?

The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library functions, control constructs or similar, so that application programmers can explicitly declare that the application follows one or more such patterns.

Keywords: skeleton, template, pattern, archetype, higher order function

What is Structured Parallel Programming?

The structured approach to parallelism proposes that commonly used patterns of computation and interaction should be abstracted as parameterisable library functions, control constructs or similar, so that application programmers can explicitly declare that the application follows one or more such patterns.

Keywords: skeleton, template, pattern, archetype, higher order function

This matters because it gives a tractable handle on the issues which make correct, efficient parallel programming hard.

Why Parallel Programming is Hard

Why Parallel Programming is Hard

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

Why Parallel Programming is Hard

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

We must maintain high efficiency in practice (not just big O).

Why Parallel Programming is Hard

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

We must maintain high efficiency in practice (not just big O).

Expressing and optimising interaction is confusing.

Does Structured Parallel Programming Help?

Does Structured Parallel Programming Help?

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

Does Structured Parallel Programming Help?

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

No. Skeletons help us express algorithms but we still have to devise them.

Does Structured Parallel Programming Help?

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

No. Skeletons help us express algorithms but we still have to devise them.

We must maintain high efficiency in practice (not just big O).

Does Structured Parallel Programming Help?

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

No. Skeletons help us express algorithms but we still have to devise them.

We must maintain high efficiency in practice (not just big O).

Yes. The skeleton implementation knows the interaction pattern in advance, and exploits this knowledge at a low level.

Does Structured Parallel Programming Help?

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

No. Skeletons help us express algorithms but we still have to devise them.

We must maintain high efficiency in practice (not just big O).

Yes. The skeleton implementation knows the interaction pattern in advance, and exploits this knowledge at a low level.

Expressing and optimising interaction is confusing.

Does Structured Parallel Programming Help?

Devising correct, efficient sequential algorithms is hard already. Introducing efficient parallelism adds an extra conceptual dimension.

No. Skeletons help us express algorithms but we still have to devise them.

We must maintain high efficiency in practice (not just big O).

Yes. The skeleton implementation knows the interaction pattern in advance, and exploits this knowledge at a low level.

Expressing and optimising interaction is confusing.

Yes. The pattern abstracted by the skeleton hides all the interaction.

How Performance Optimisations Work

Well known, tried and trusted performance optimisations typically exploit information about the spatial and temporal context in which indvidual operations are performed.

Cache Optimisations

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++)
       for (k=0; k<N; k++)
           c[i][j] += a[i][k]*b[k][j];
```

Cache Optimisations

```
for (jj=0; jj<N; jj=jj+B)
   for (kk=0; kk<N; kk=kk+B)
      for (i=0; i<N; i++)
         for (j=jj; j < jj+B; j++) {
            pa = &a[i][kk]; pb = &b[kk][j];
            temp = (*pa++)*(*pb);
            for (k=kk+1; k < kk+B; k++) {
              pb = pb+N;
              temp += (*pa++)*(*pb);
            }
            c[i][j] += temp;
      }
```

Branch Prediction

```
while (a[i] < b[a[i]]) {
    c[j+i] += b[j];
    if (c[k] < 100) {
        k++;
        b[j] = 0;
    }
    i++;
}
```

Performance improved by knowing whether branches are "usually" taken.

How Performance Optimisations Work

Well known, tried and trusted performance optimisations typically exploit information about the spatial and temporal context in which indvidual operations are performed.

How Performance Optimisations Work

Well known, tried and trusted performance optimisations typically exploit information about the spatial and temporal context in which indvidual operations are performed.

The use of structured parallel programming techniques provides information about future interactions, sometimes for the entire execution, in context.

How Performance Optimisations Work

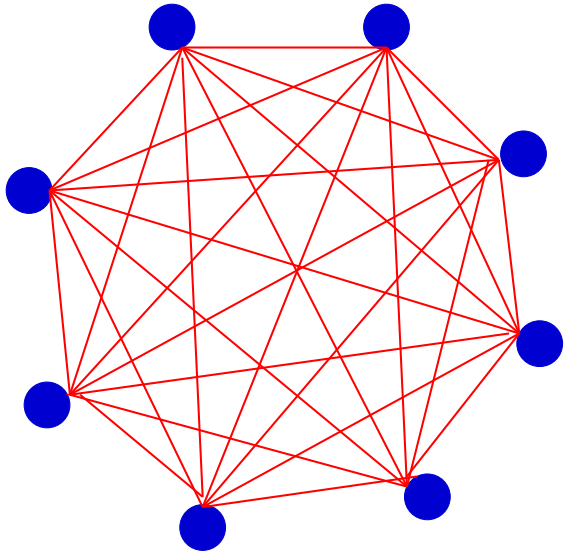
Well known, tried and trusted performance optimisations typically exploit information about the spatial and temporal context in which indvidual operations are performed.

The use of structured parallel programming techniques provides information about future interactions, sometimes for the entire execution, in context.

Structured Parallelism lets us tell the system what will happen next.

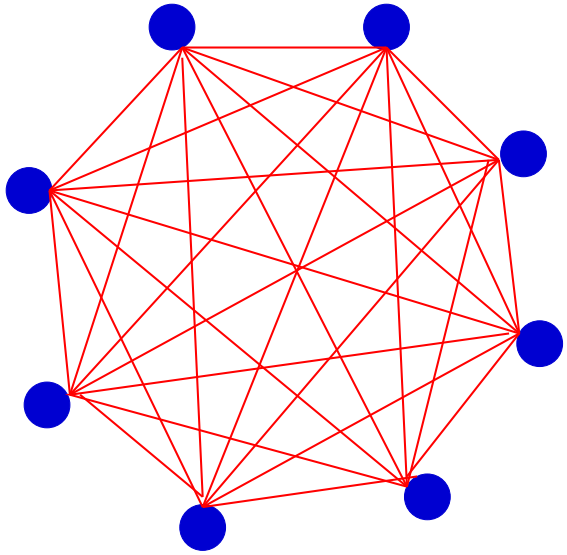
Parallel Performance Optimisations

Consider a simple "iterated all-pairs" structure



Parallel Performance Optimisations

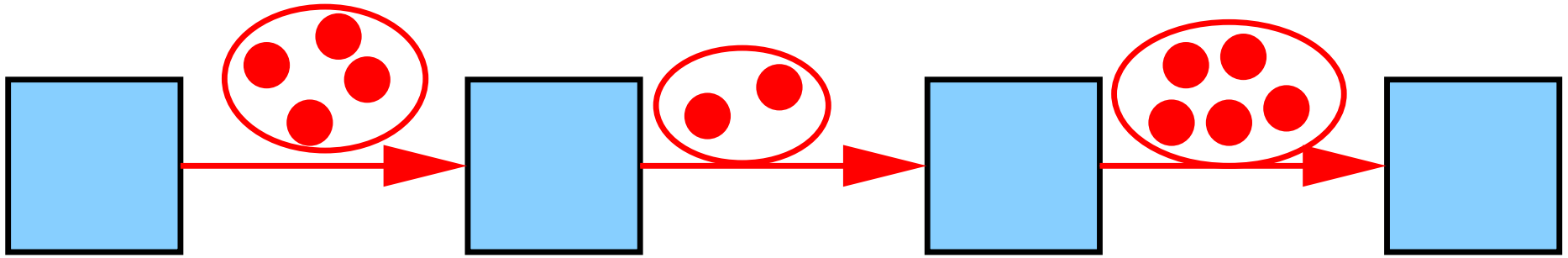
Consider a simple "iterated all-pairs" structure



Careful shared memory scheduling is essential.

Parallel Performance Optimisations

In a simple pipeline, agglomeration of messages can be crucial.



Parallel Performance Optimisations

It will be difficult (often impossible?) to recognise that what we have is an "all-pairs" or "pipeline" computation from the equivalent unstructured source.

Parallel Performance Optimisations

It will be difficult (often impossible?) to recognise that what we have is an "all-pairs" or "pipeline" computation from the equivalent unstructured source.

Structured Parallelism matters because it allows us to give the system this information (and as a side-effect, simplifies our programming task).

Higher Level Performance Optimisation

Higher Level Performance Optimisation

Performance programming is also about choosing the right algorithm, and being sure that it is correct.

Higher Level Performance Optimisation

Performance programming is also about choosing the right algorithm, and being sure that it is correct.

Designing algorithms with structured concepts in mind allows us to benefit from high level algorithm restructuring techniques.

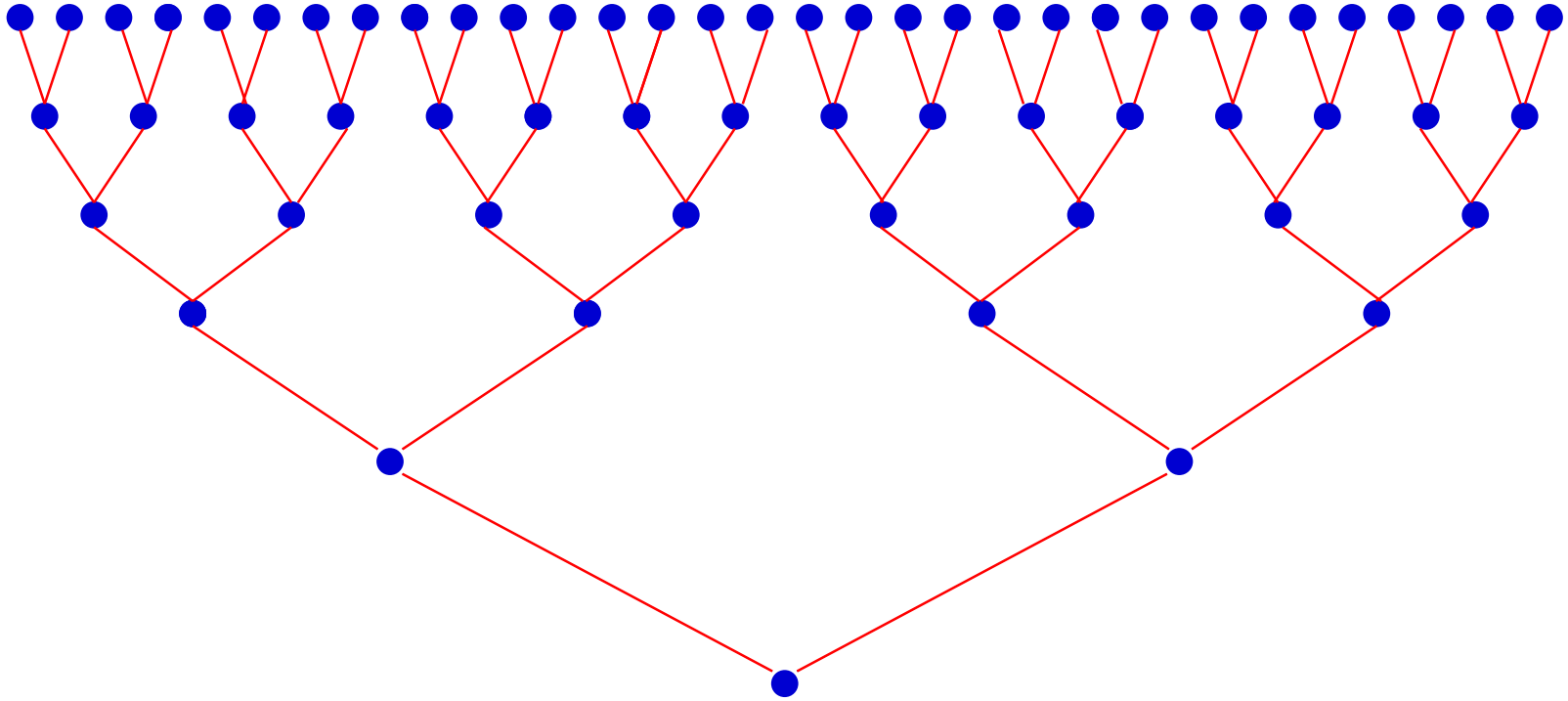
Higher Level Performance Optimisation

Performance programming is also about choosing the right algorithm, and being sure that it is correct.

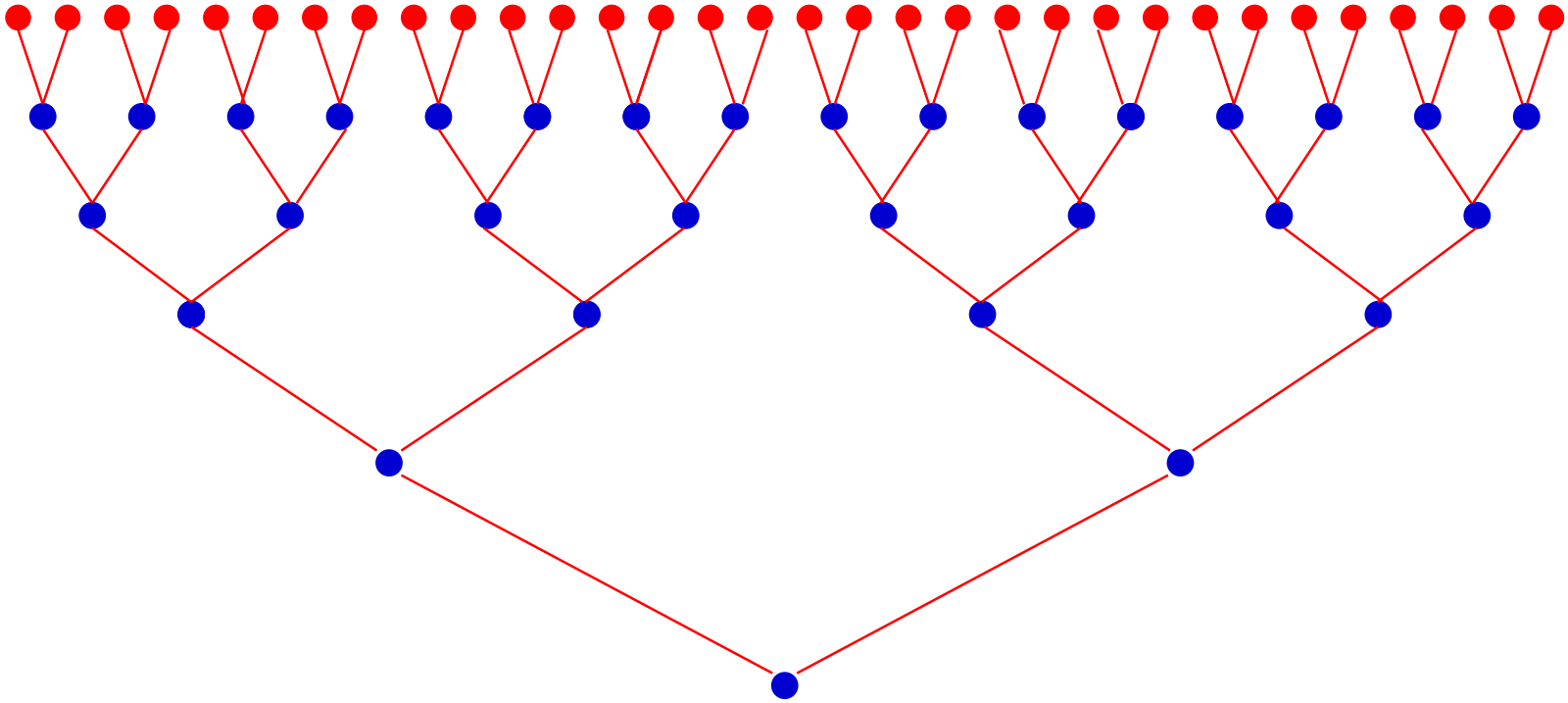
Designing algorithms with structured concepts in mind allows us to benefit from high level algorithm restructuring techniques.

The coarse grain, collective nature of skeletons allows substantial transformations to be made with a small number of steps (in contrast to the same effect achieved with the corresponding unstructured primitives).

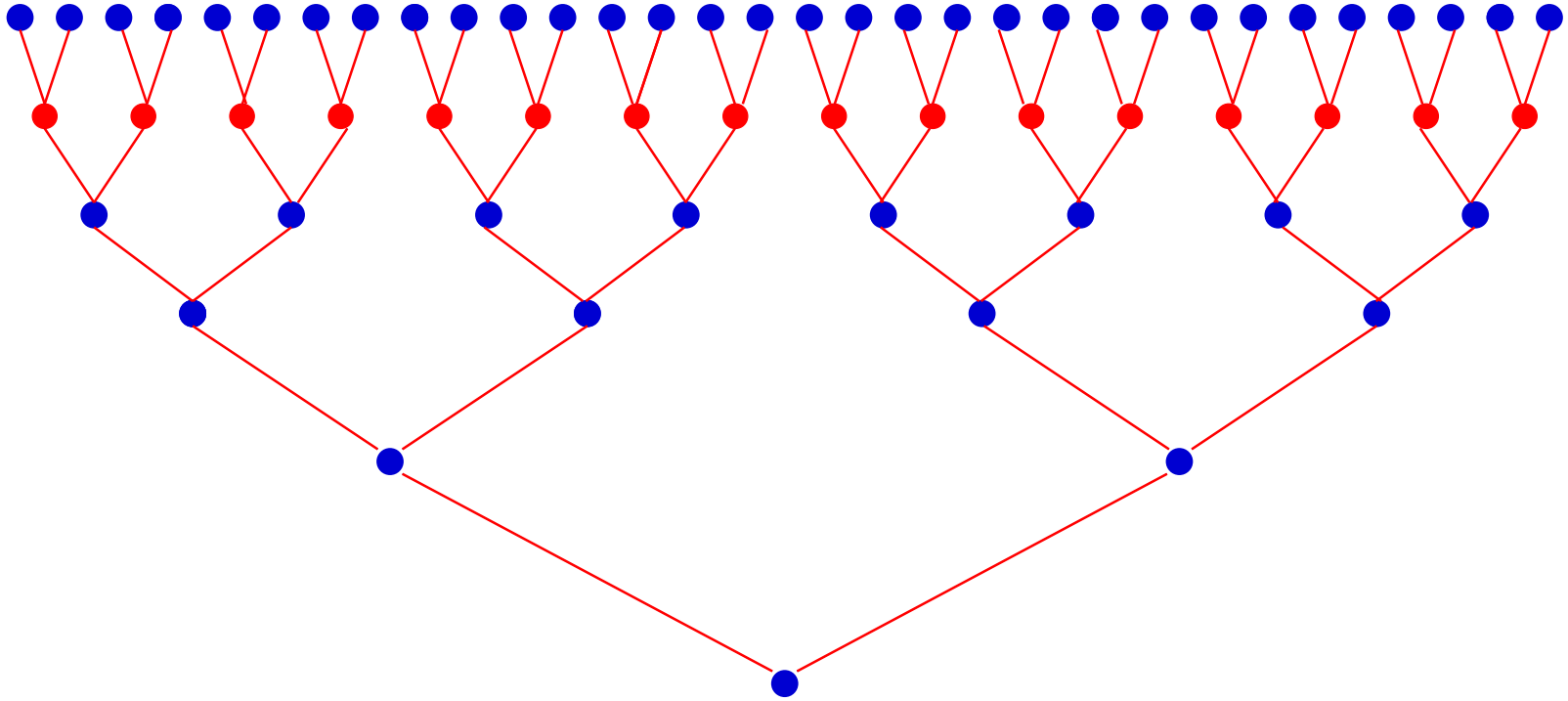
Higher Level Performance Optimisation



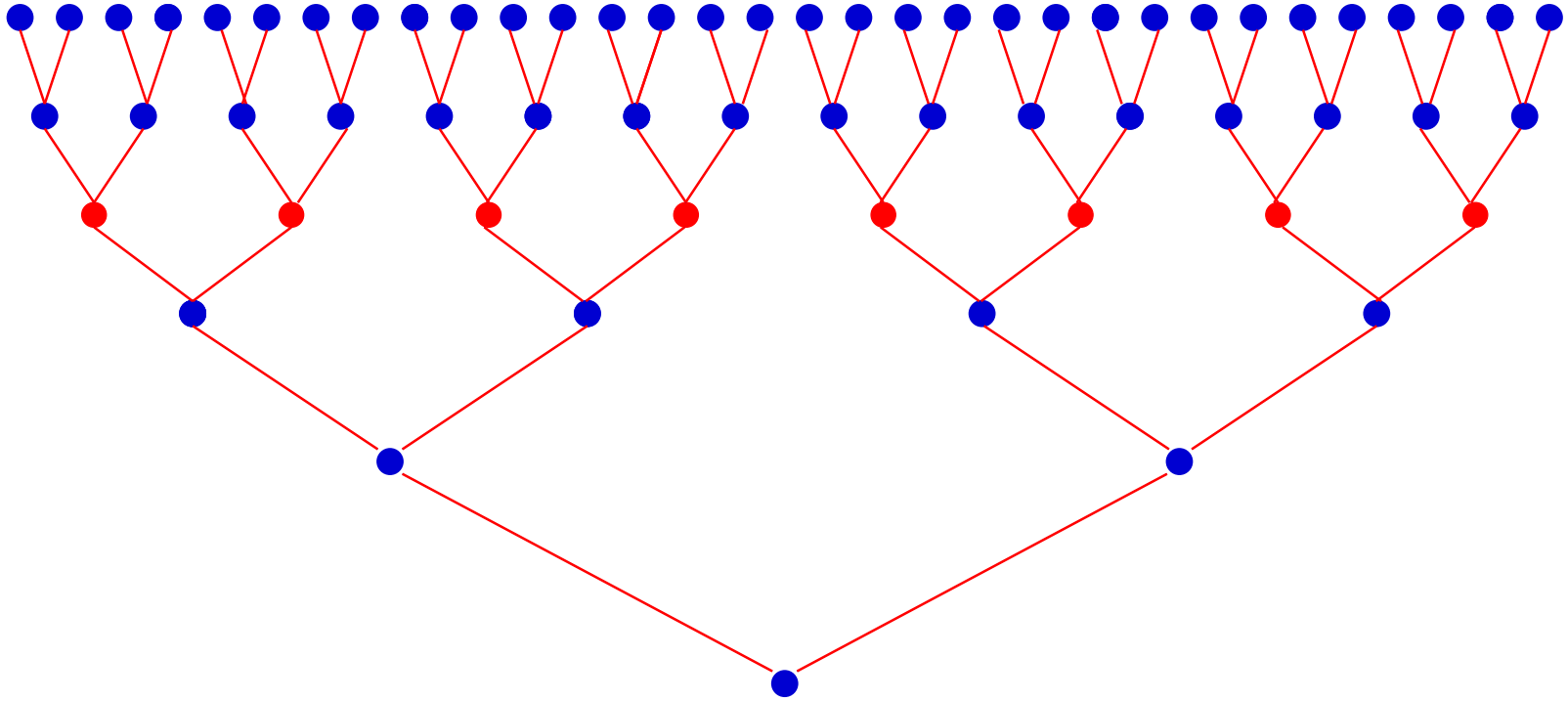
Higher Level Performance Optimisation



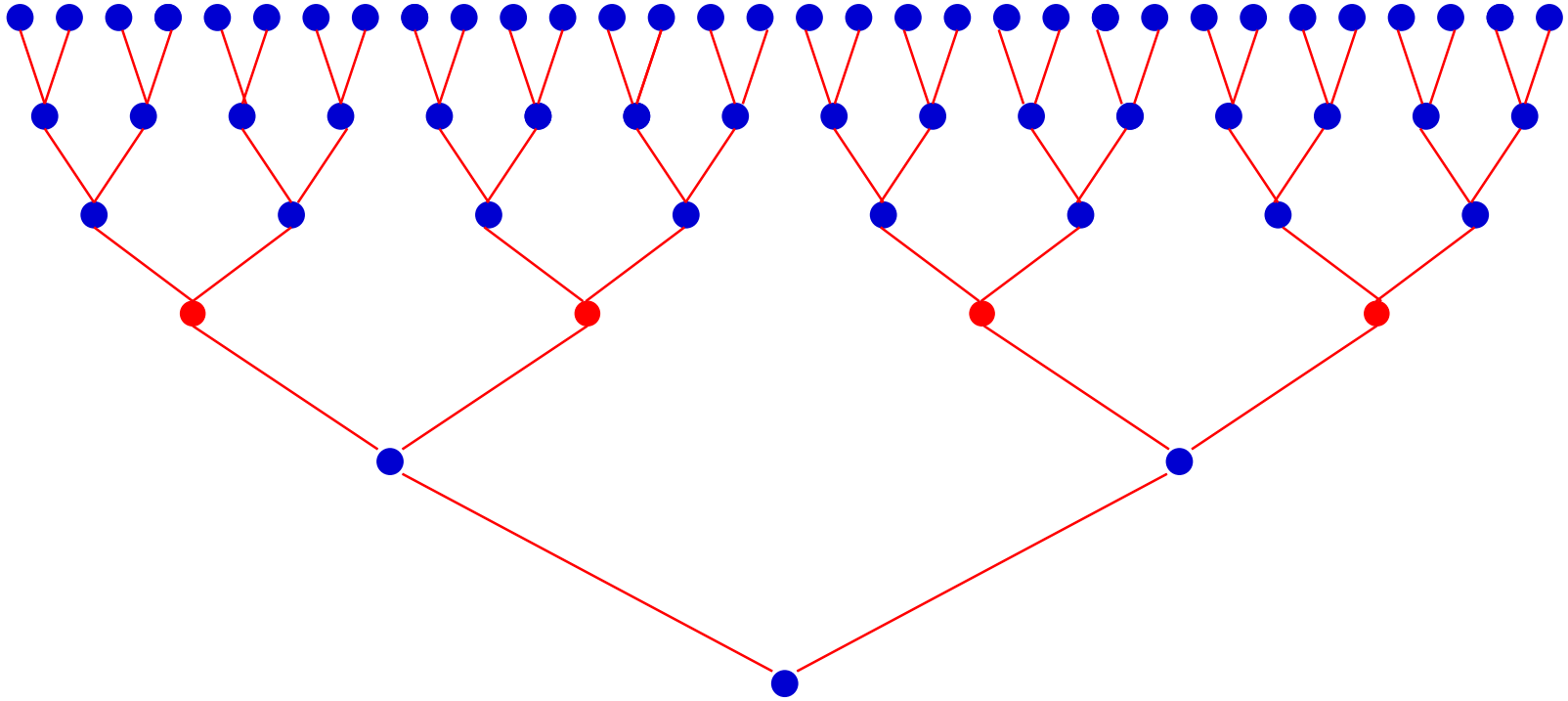
Higher Level Performance Optimisation



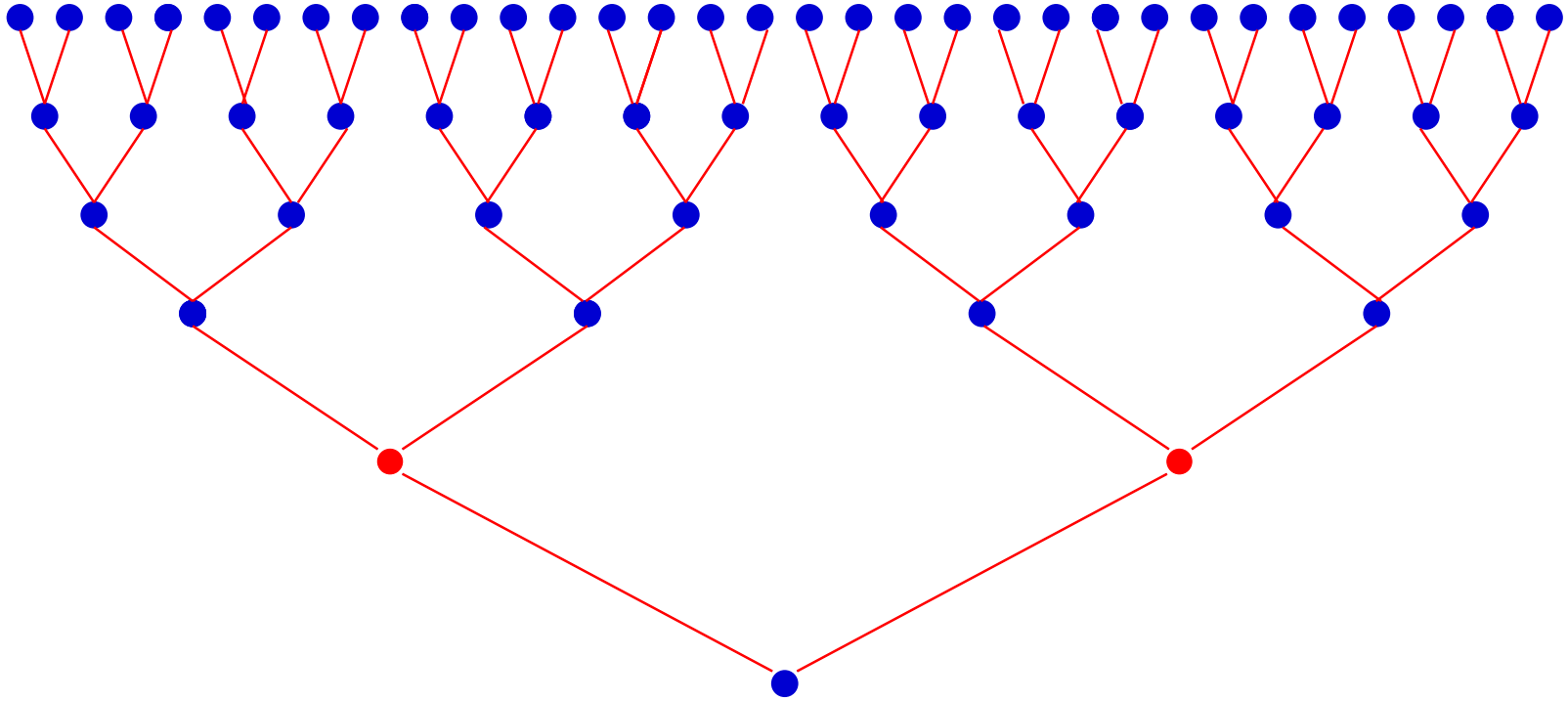
Higher Level Performance Optimisation



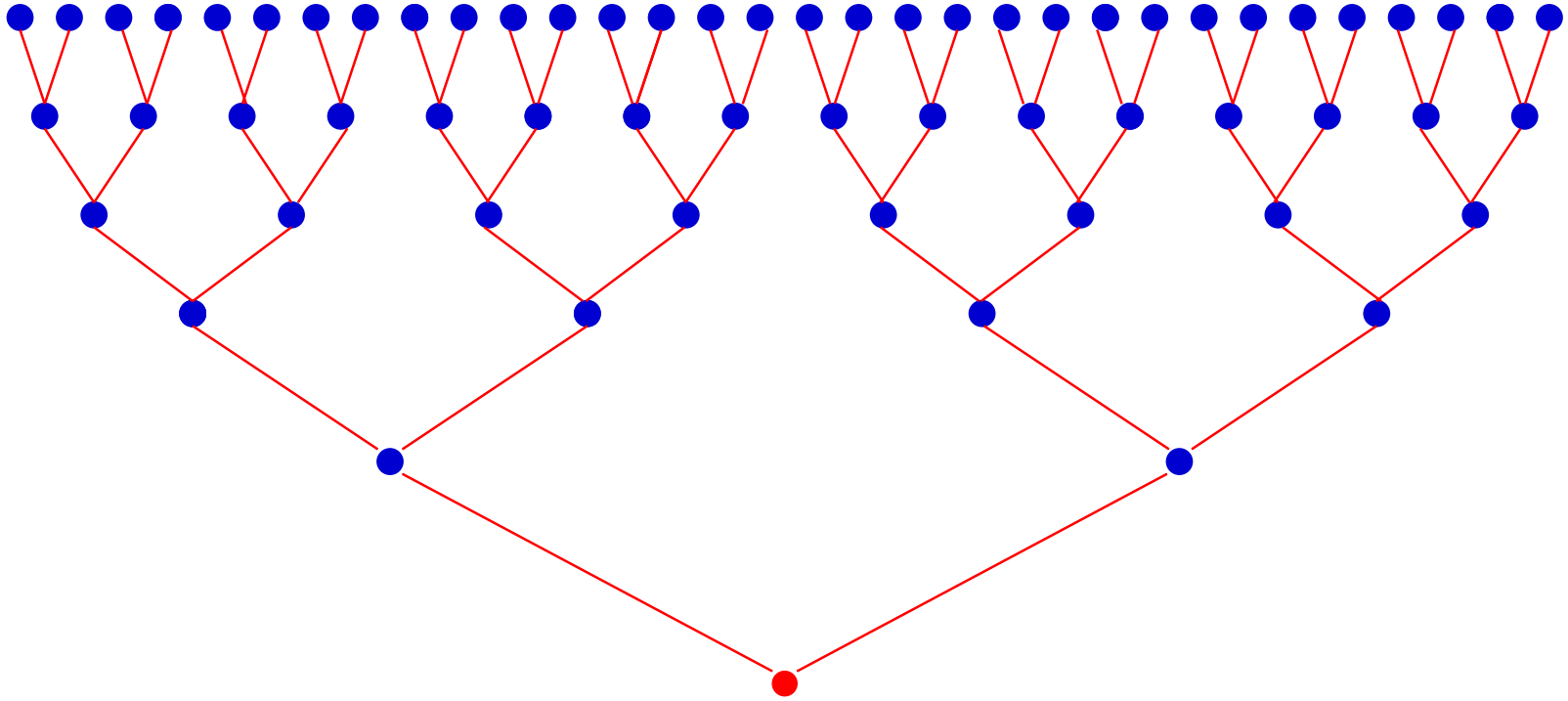
Higher Level Performance Optimisation



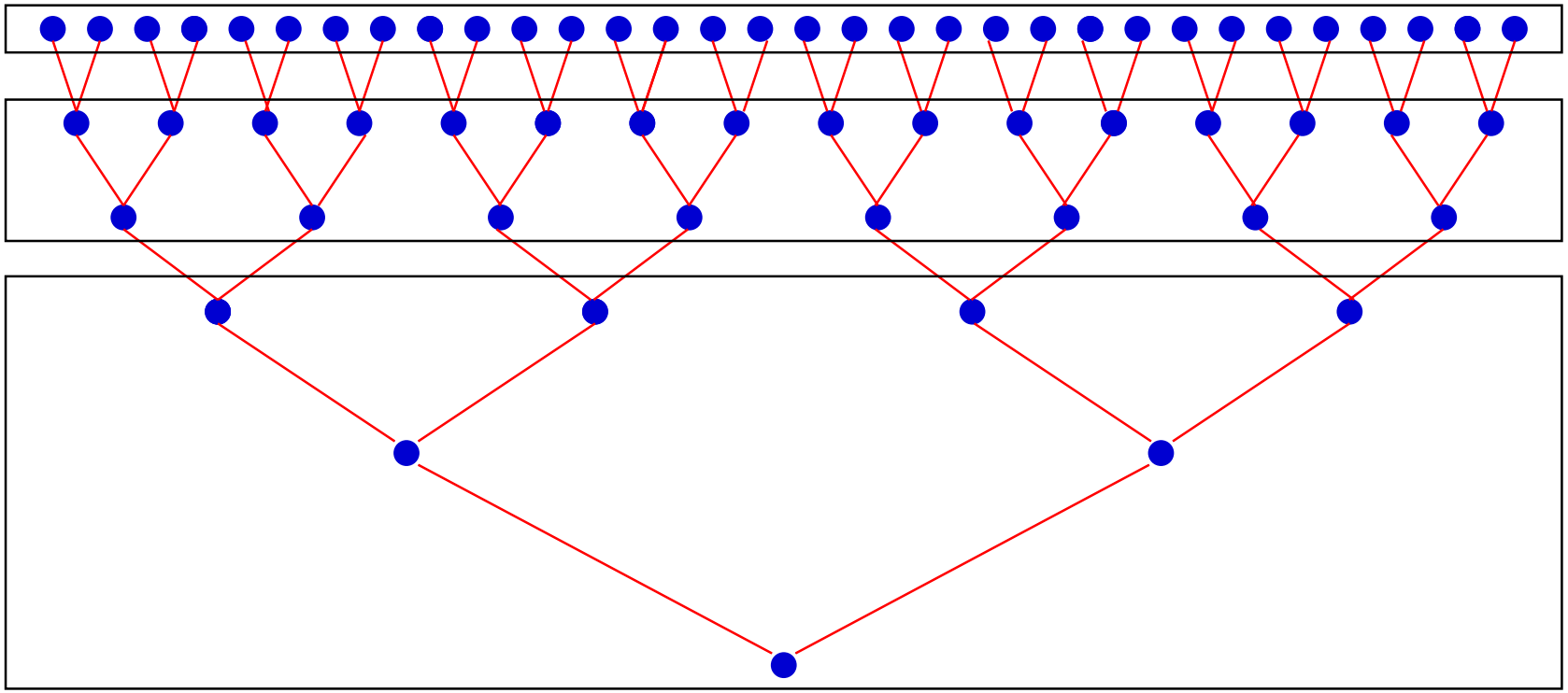
Higher Level Performance Optimisation



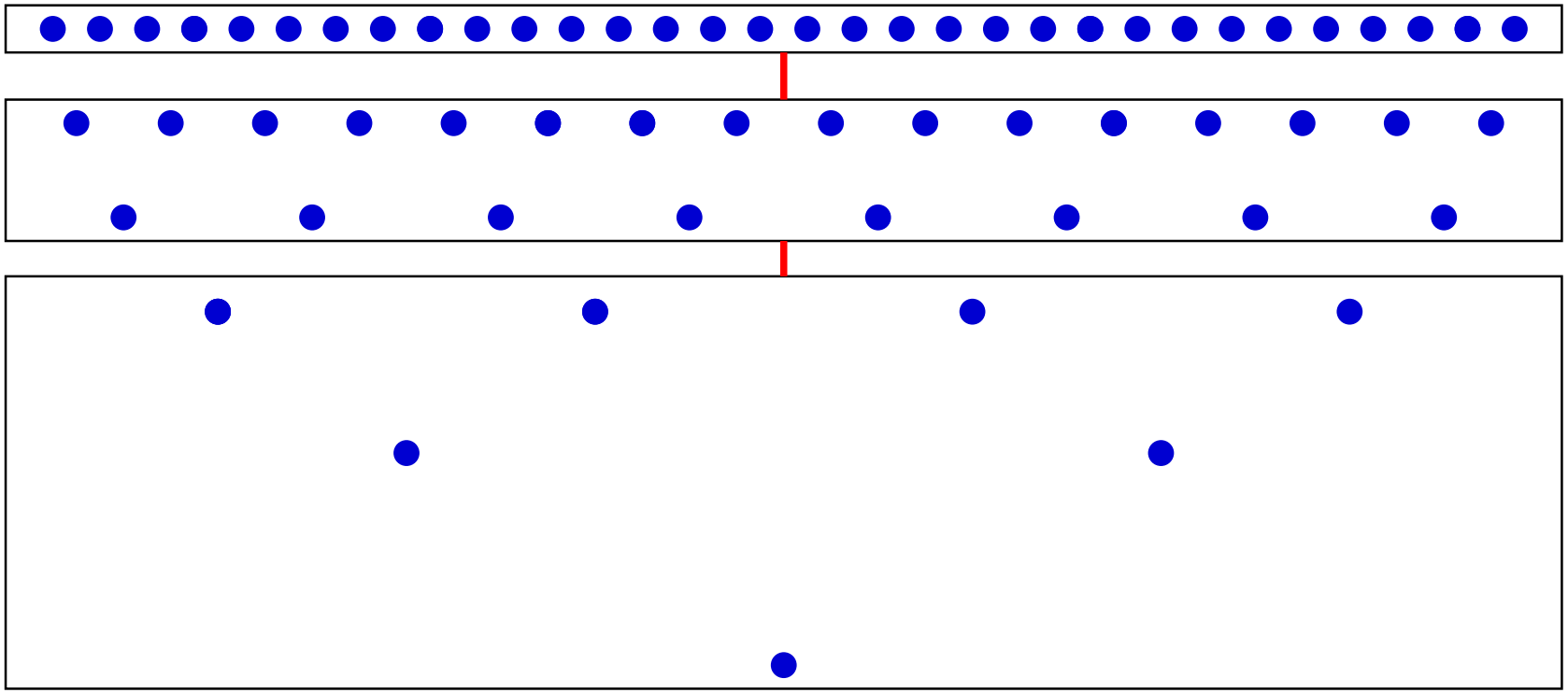
Higher Level Performance Optimisation



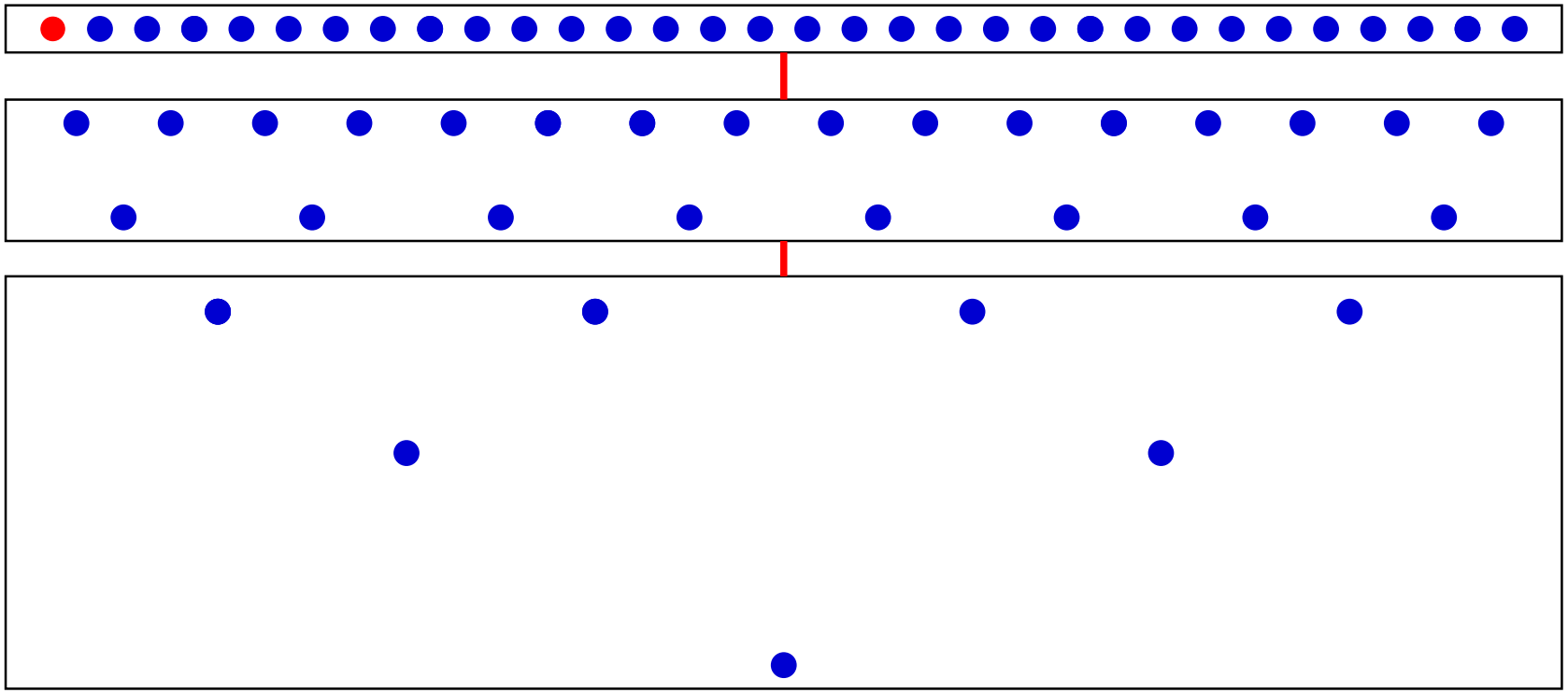
Higher Level Performance Optimisation



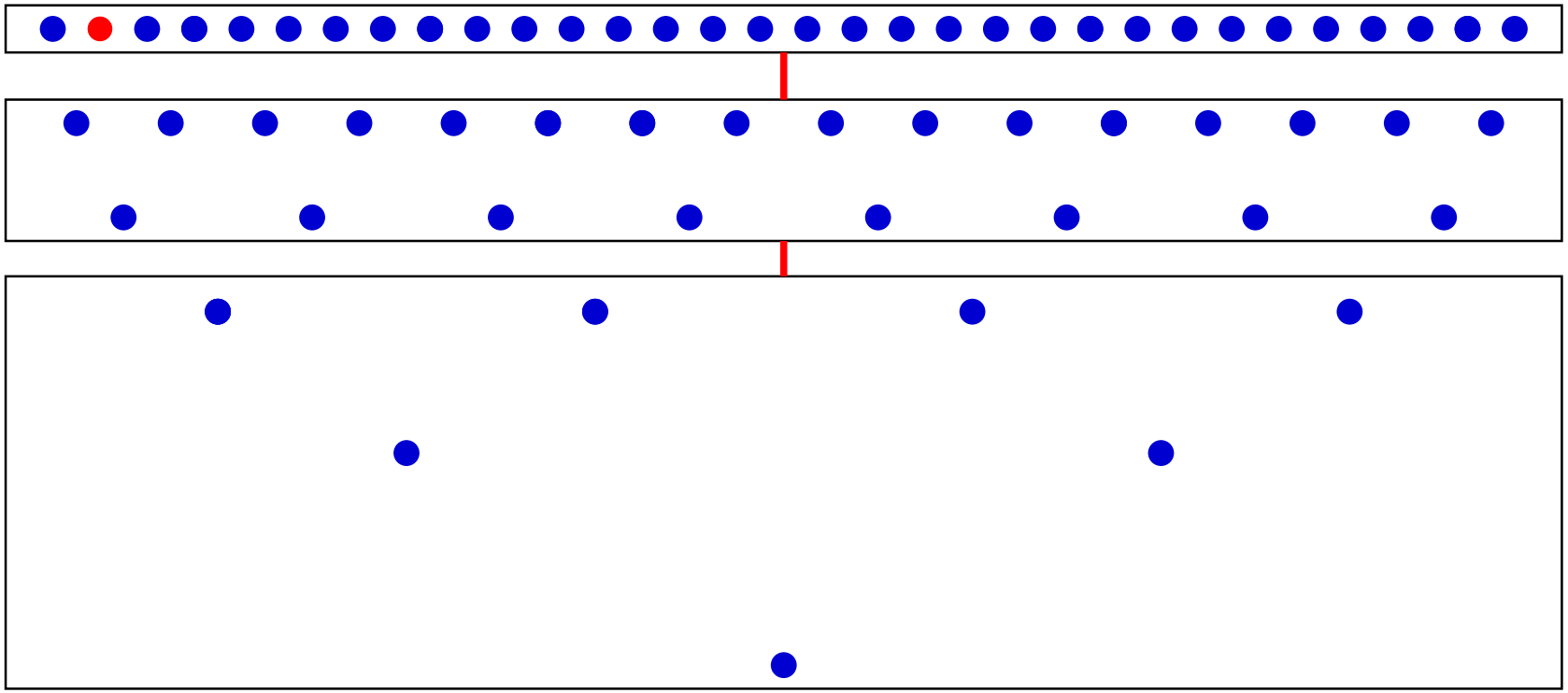
Higher Level Performance Optimisation



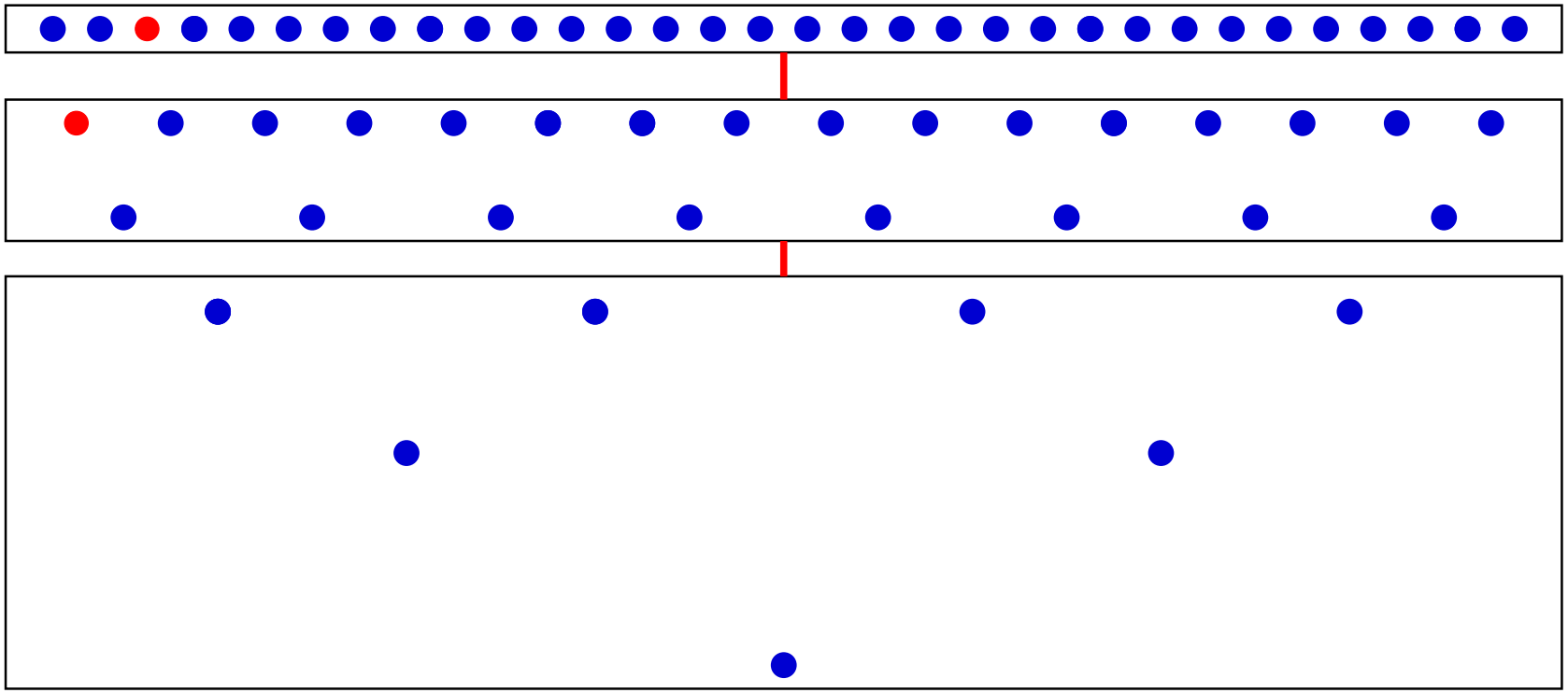
Higher Level Performance Optimisation



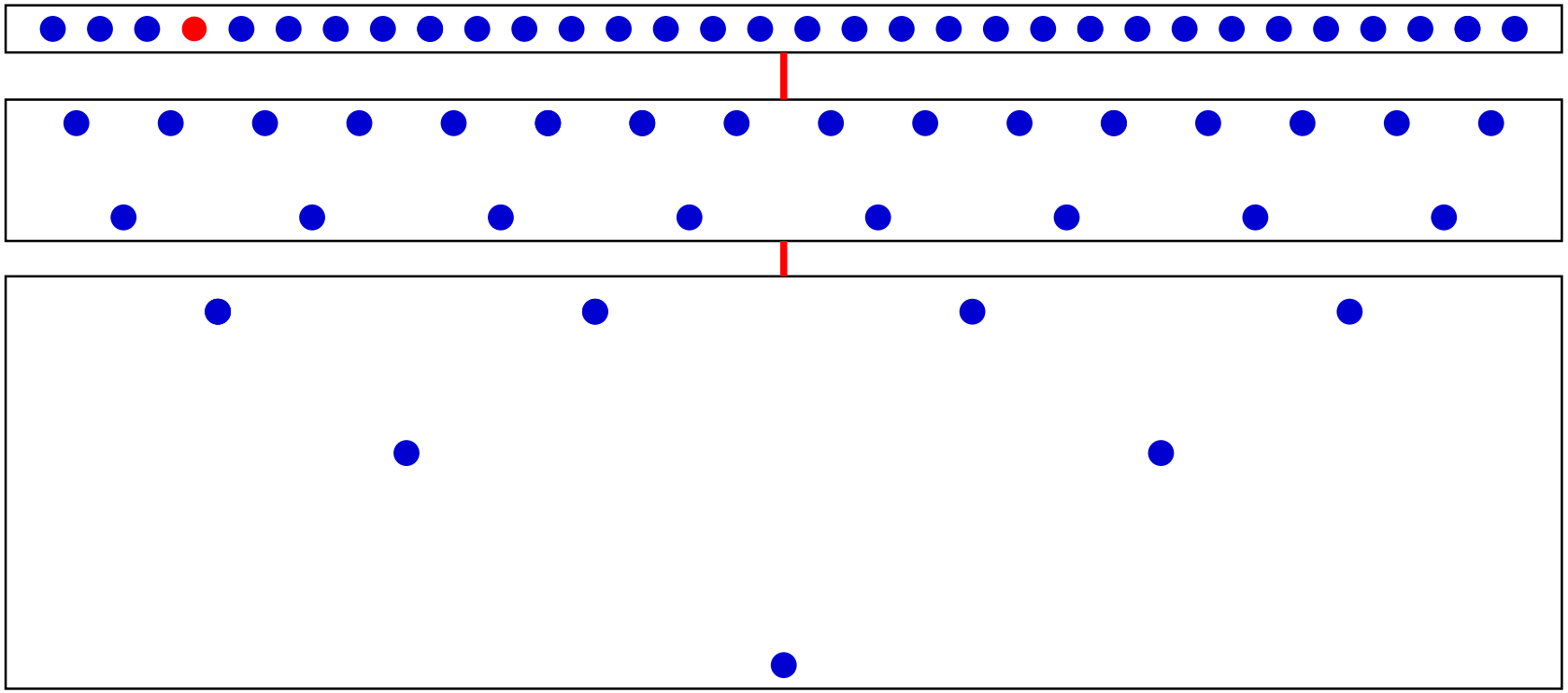
Higher Level Performance Optimisation



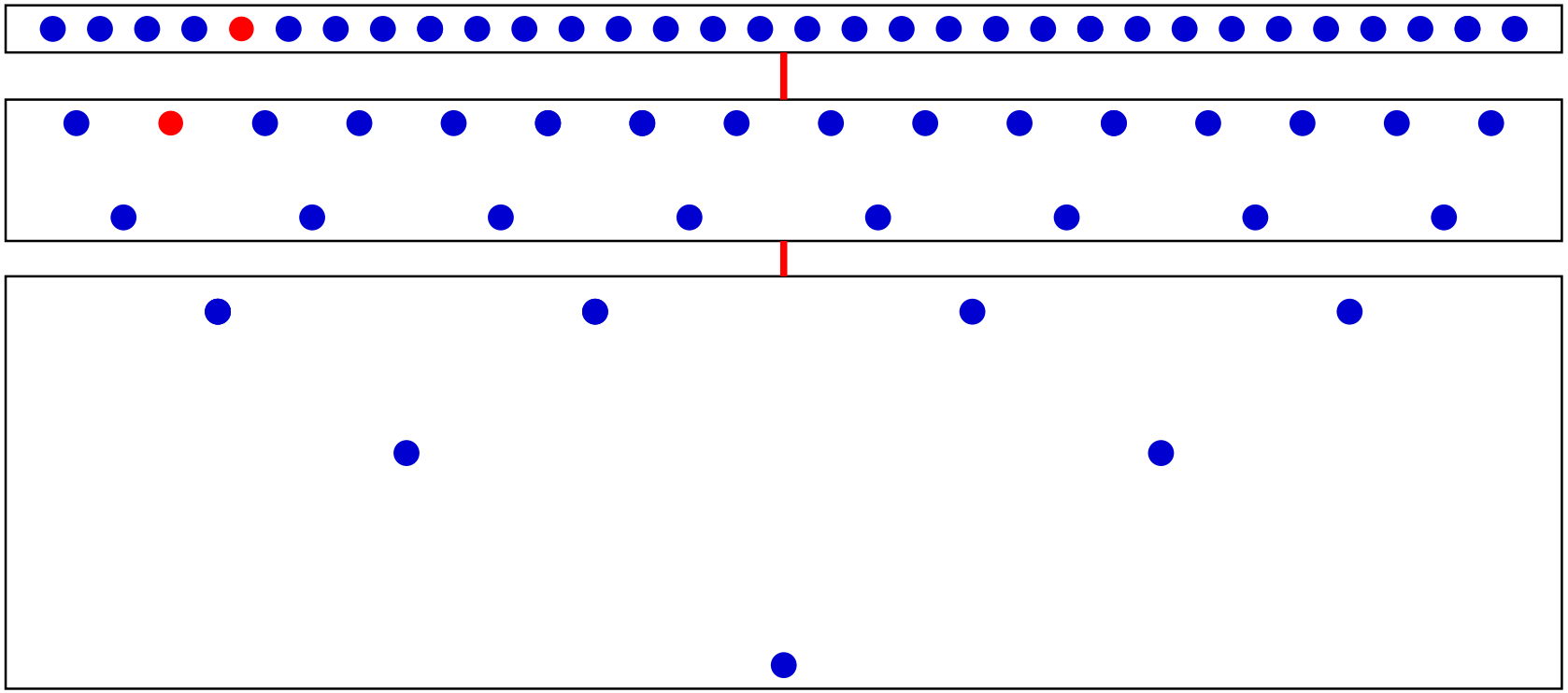
Higher Level Performance Optimisation



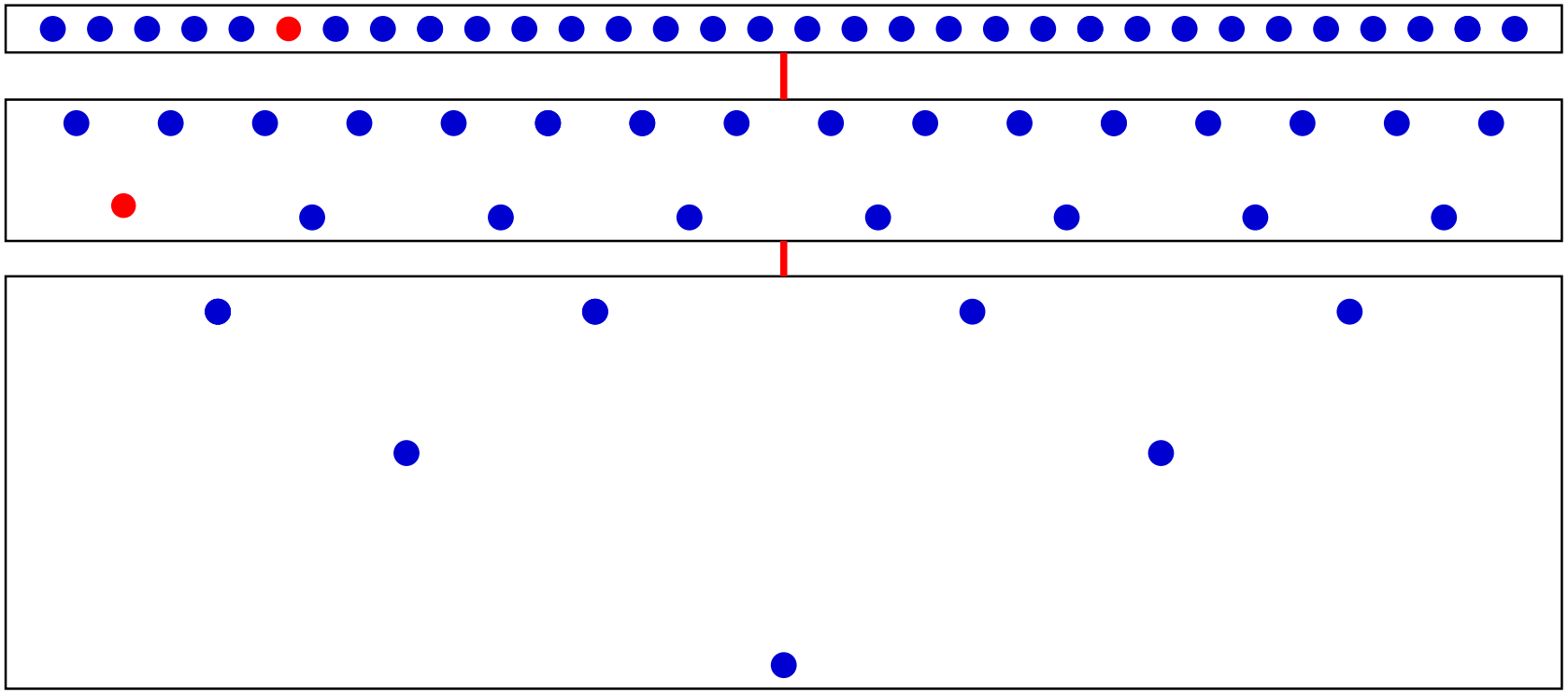
Higher Level Performance Optimisation



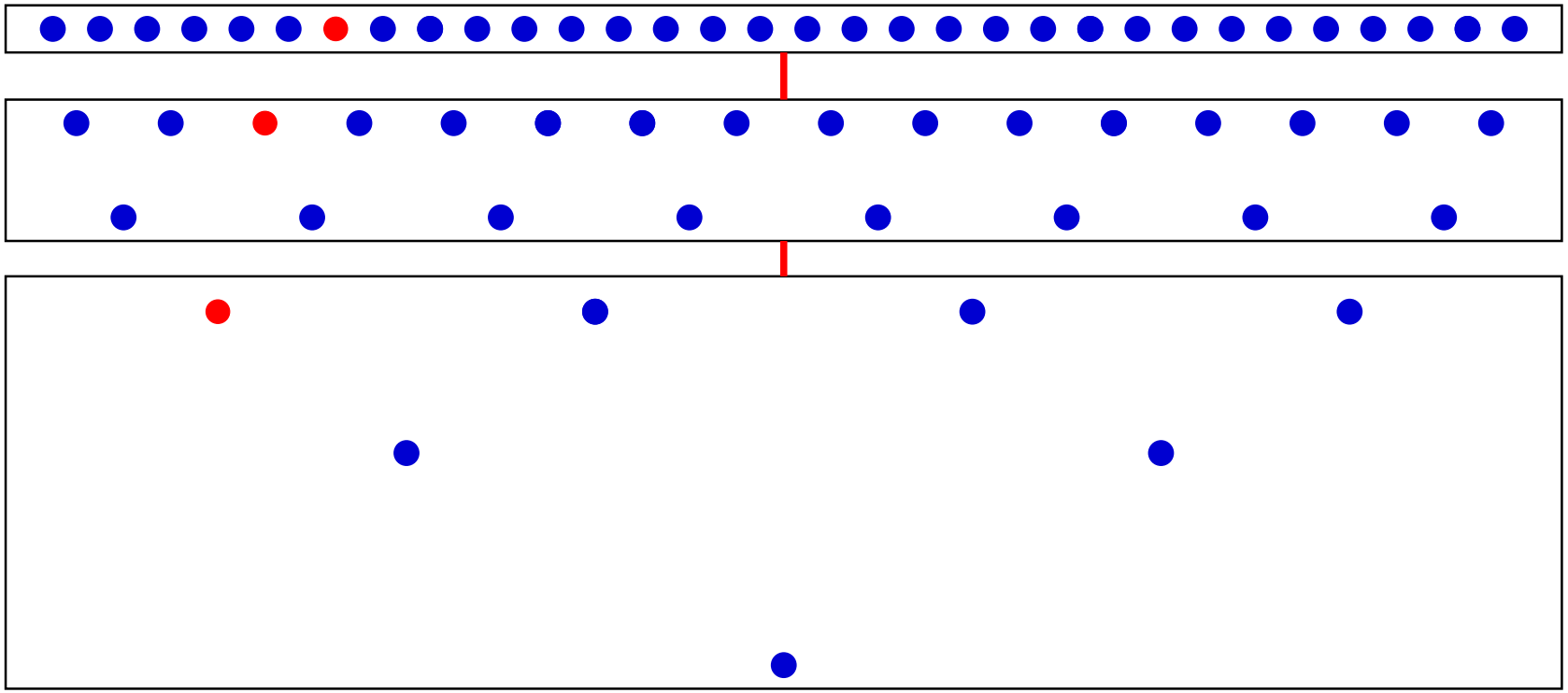
Higher Level Performance Optimisation



Higher Level Performance Optimisation



Higher Level Performance Optimisation



Higher Level Performance Optimisation

Structured Parallelism matters because it allows us to manipulate parallel algorithms at a coarse structural level.

Who cares?

Who cares?

Skeletal programming remains a fringe activity

A Pragmatic Manifesto

A Pragmatic Manifesto

1. Minimise Conceptual Disruption

A Pragmatic Manifesto

1. Minimise Conceptual Disruption

There are many ways of presenting these ideas.

A Pragmatic Manifesto

1. Minimise Conceptual Disruption

There are many ways of presenting these ideas.

Parallel programmers are happy with C/Fortran and MPI.

A Pragmatic Manifesto

1. Minimise Conceptual Disruption

There are many ways of presenting these ideas.

Parallel programmers are happy with C/Fortran and MPI.

MPI's collectives are simple skeletons, so build on this.

A Pragmatic Manifesto

2. Integrate Ad-Hoc Parallelism

A Pragmatic Manifesto

2. Integrate Ad-Hoc Parallelism

Sometimes parallelism seems inherently unstructured.

A Pragmatic Manifesto

2. Integrate Ad-Hoc Parallelism

Sometimes parallelism seems inherently unstructured.

Allow contained integration within a structured container.

A Pragmatic Manifesto

2. Integrate Ad-Hoc Parallelism

Sometimes parallelism seems inherently unstructured.

Allow contained integration within a structured container.

Don't overconstrain.

A Pragmatic Manifesto

3. Accommodate Diversity

A Pragmatic Manifesto

3. Accommodate Diversity

"Well-known" concepts are quite slippery.

A Pragmatic Manifesto

3. Accommodate Diversity

"Well-known" concepts are quite slippery.

Pipeline stage as function or stage as process?

A Pragmatic Manifesto

3. Accommodate Diversity

"Well-known" concepts are quite slippery.

Pipeline stage as function or stage as process?

Pipeline stage "one-for-one" or arbitrary?

A Pragmatic Manifesto

3. Accommodate Diversity

"Well-known" concepts are quite slippery.

Pipeline stage as function or stage as process?

Pipeline stage "one-for-one" or arbitrary?

Implicit or explicit farmer?

A Pragmatic Manifesto

3. Accommodate Diversity

"Well-known" concepts are quite slippery.

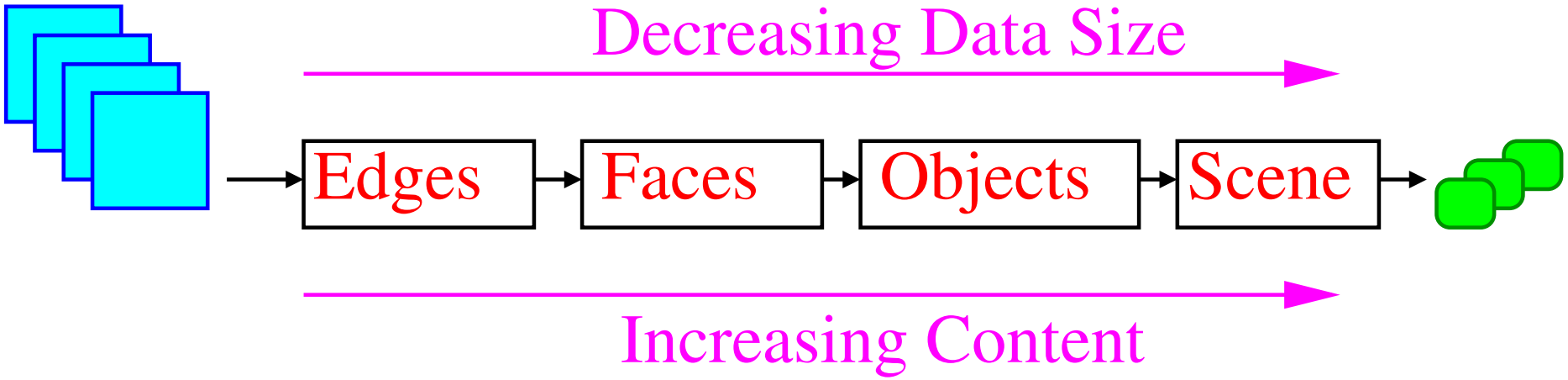
Pipeline stage as function or stage as process?

Pipeline stage "one-for-one" or arbitrary?

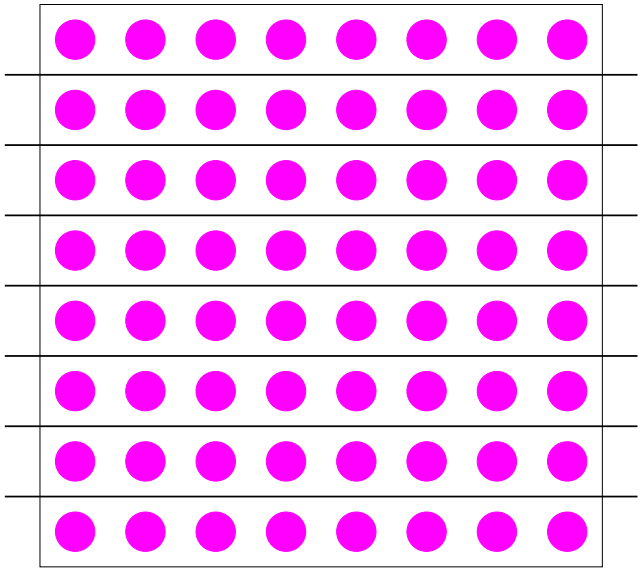
Implicit or explicit farmer?

Don't overconstrain.

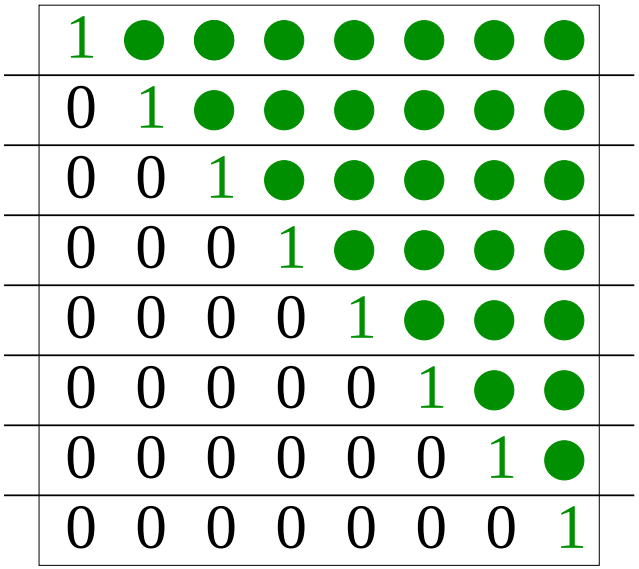
A Vanilla Pipeline - Image Processing



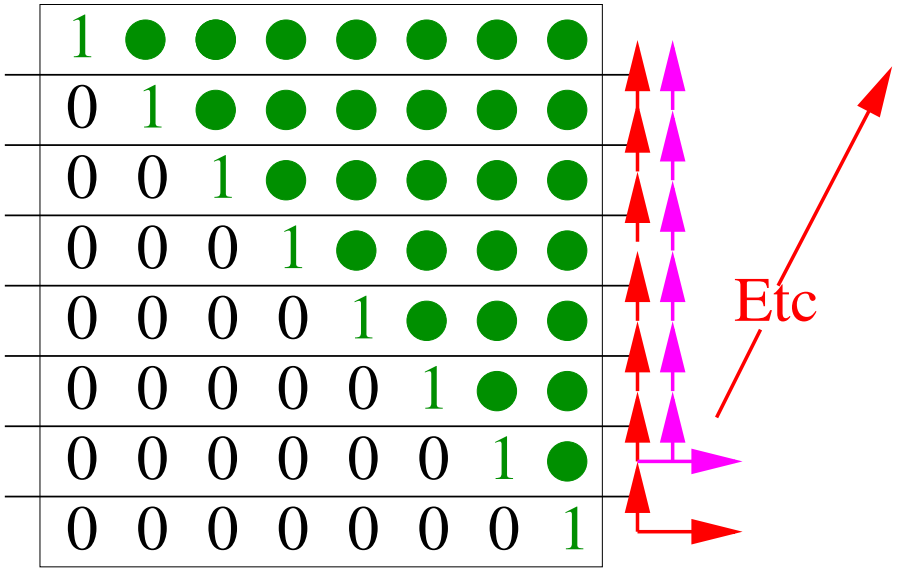
An Exotic Pipeline - Gaussian Elimination



An Exotic Pipeline - Gaussian Elimination



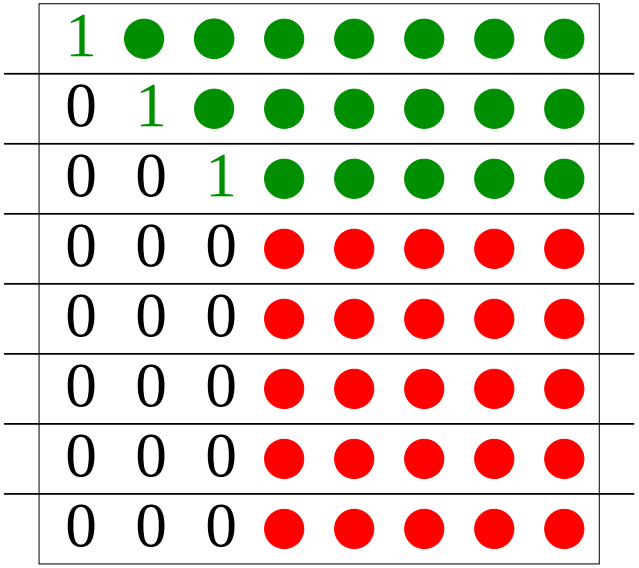
An Exotic Pipeline - Gaussian Elimination



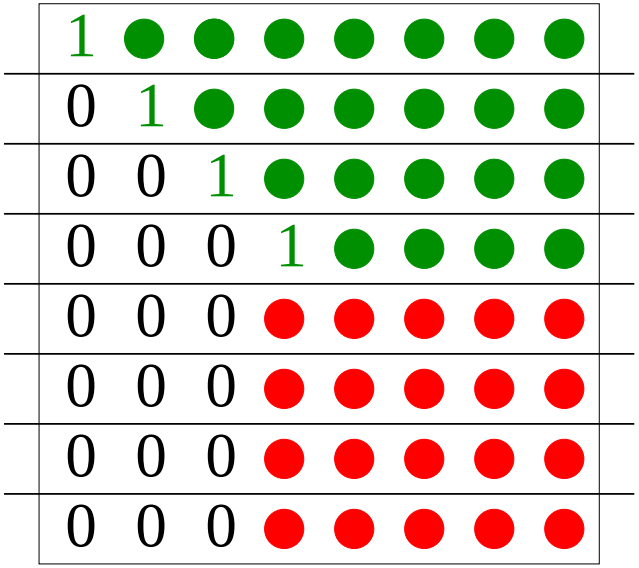
Elimination Phase

```
for (each row in sequence) {

     normalise this ‘‘pivot’’ row;

     broadcast result to subsequent rows;

     for (each subsequent row in parallel) {

         eliminate one column using broadcast row
     }
}
```

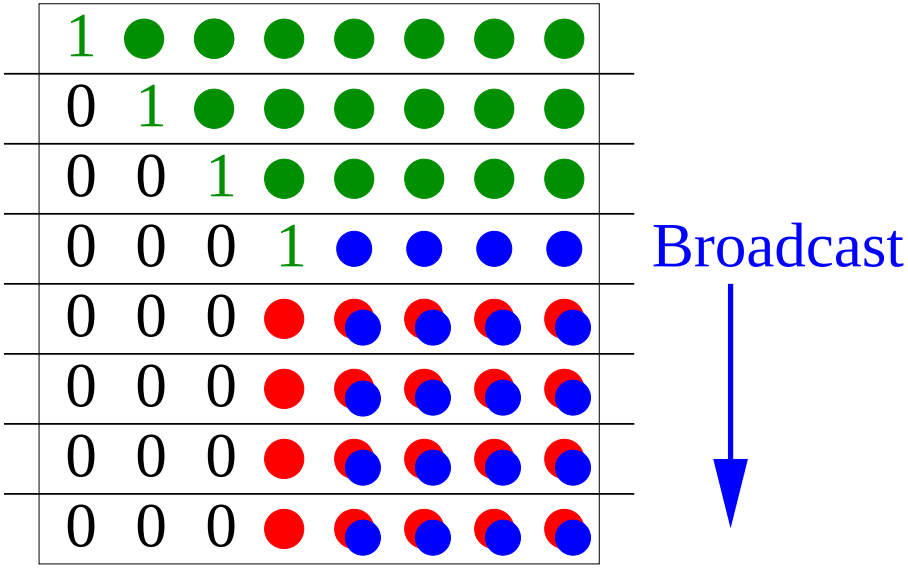
Elimination Phase



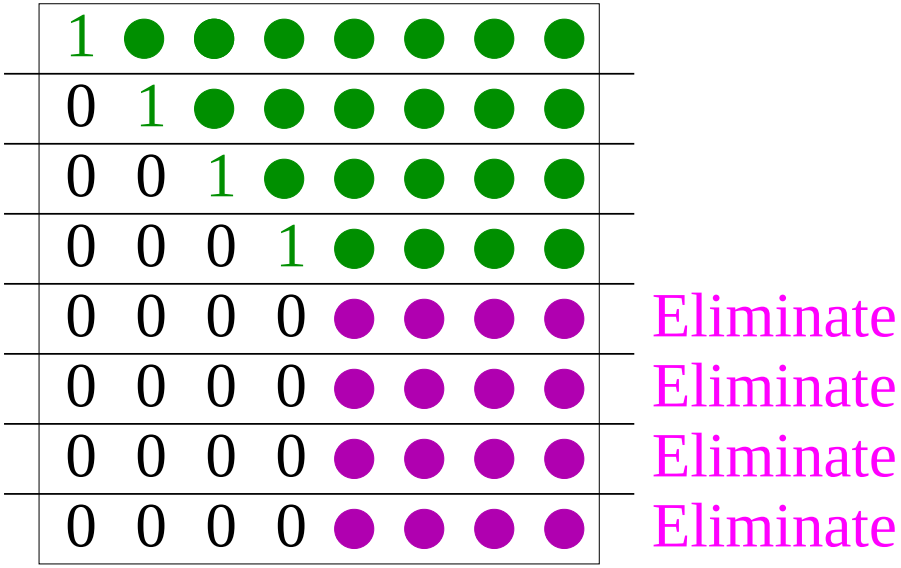
Elimination Phase



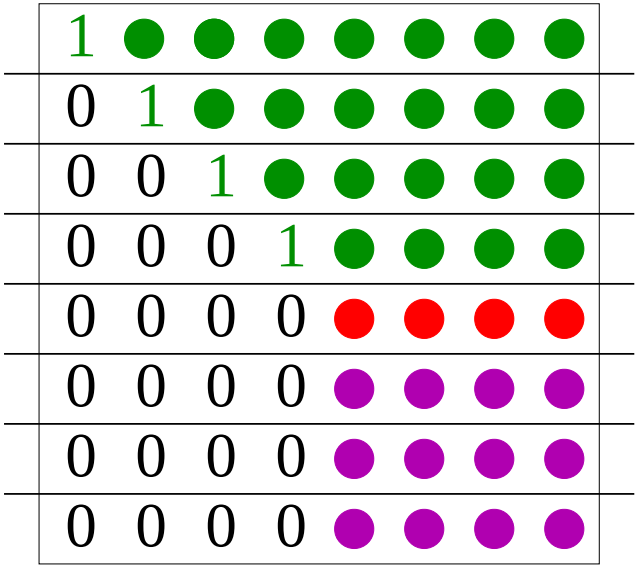
Elimination Phase



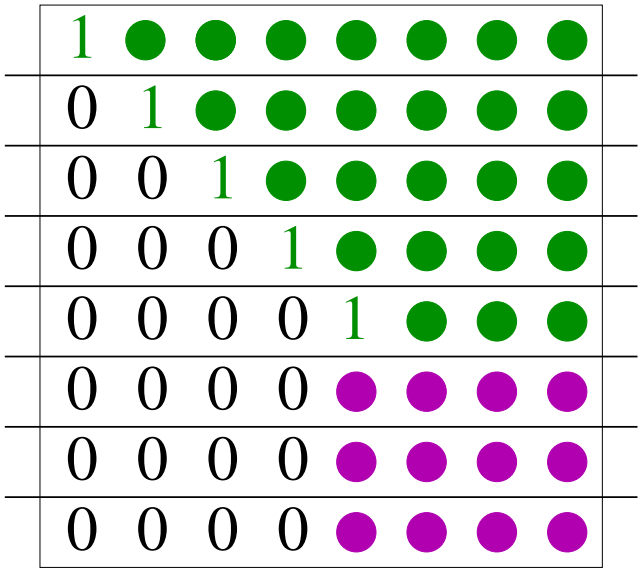
Elimination Phase



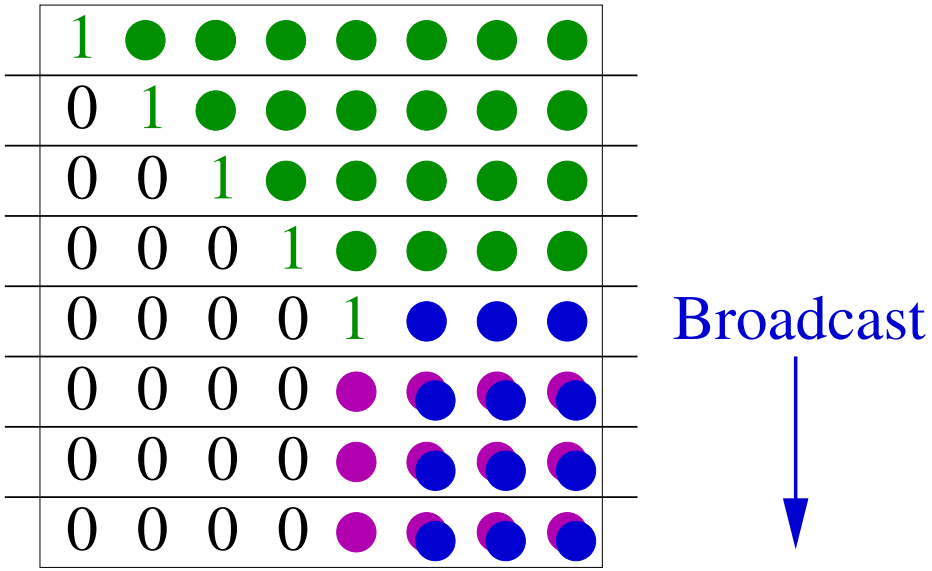
Elimination Phase



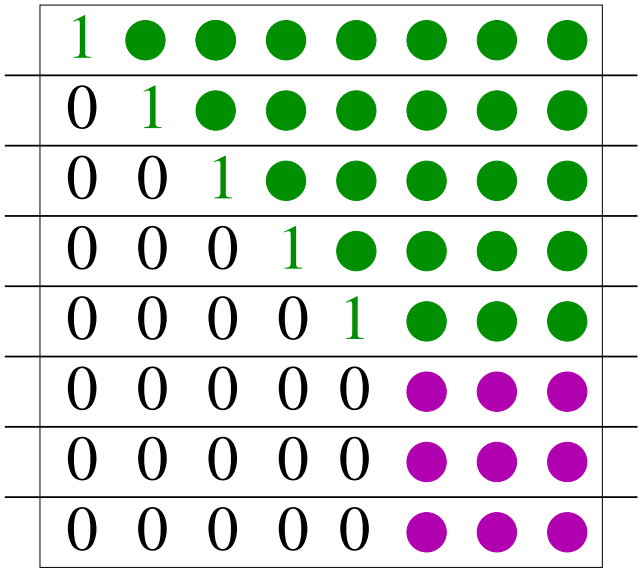
Elimination Phase



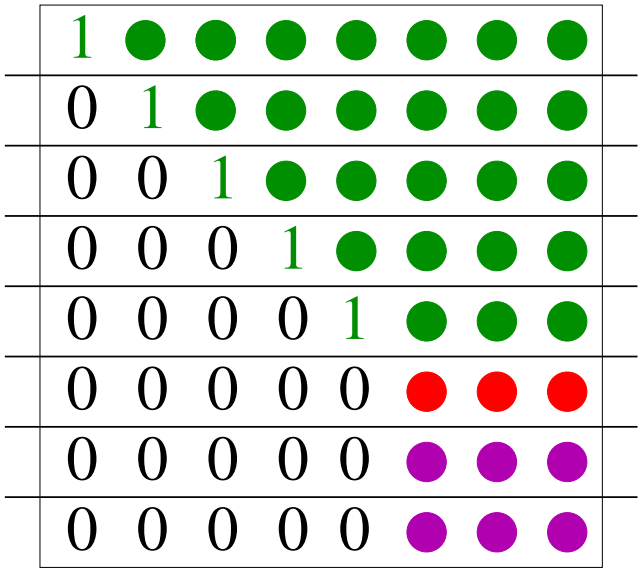
Elimination Phase



Elimination Phase



Elimination Phase



Pipelined Version

Pipelined Version

Textbook improvement interleaves broadcast and elimination phases.

Pipelined Version

Textbook improvement interleaves broadcast and elimination phases.

Processors

- participate in broadcast
- begin elimination immediately (before broadcast completes elsewhere)
- iterations become pipelined

Pipelined Version

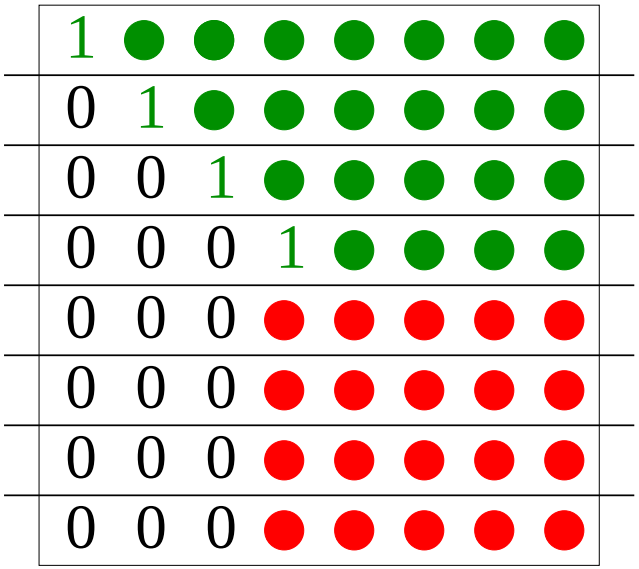
Textbook improvement interleaves broadcast and elimination phases.

Processors

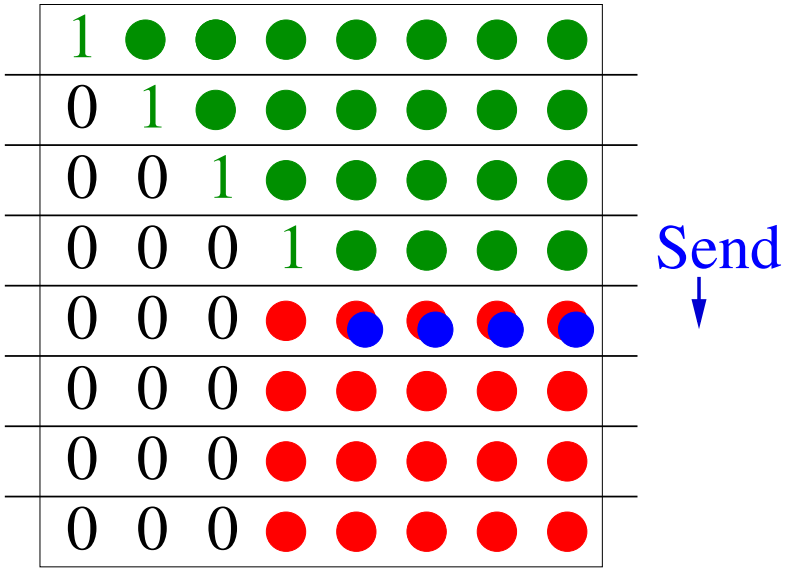
- participate in broadcast
- begin elimination immediately (before broadcast completes elsewhere)
- iterations become pipelined

Each iteration would be slower independently, but pipelining across iterations produces an overall gain.

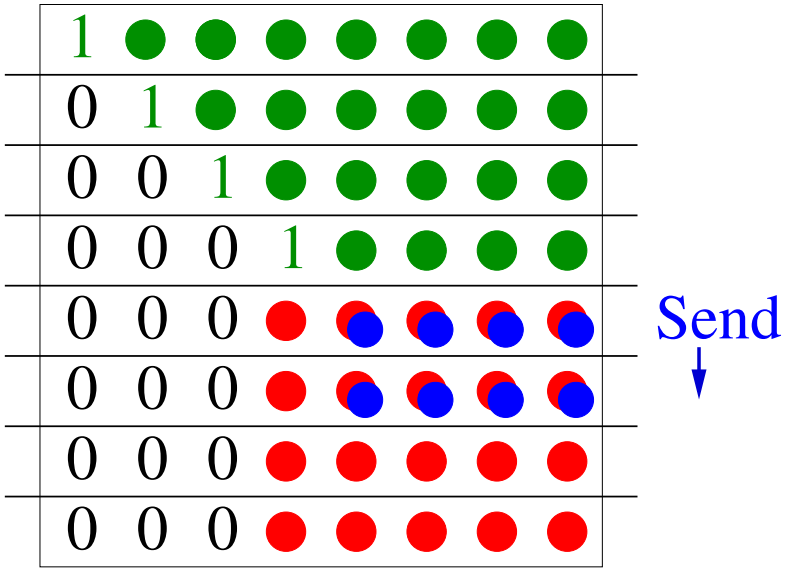
Pipelined Version



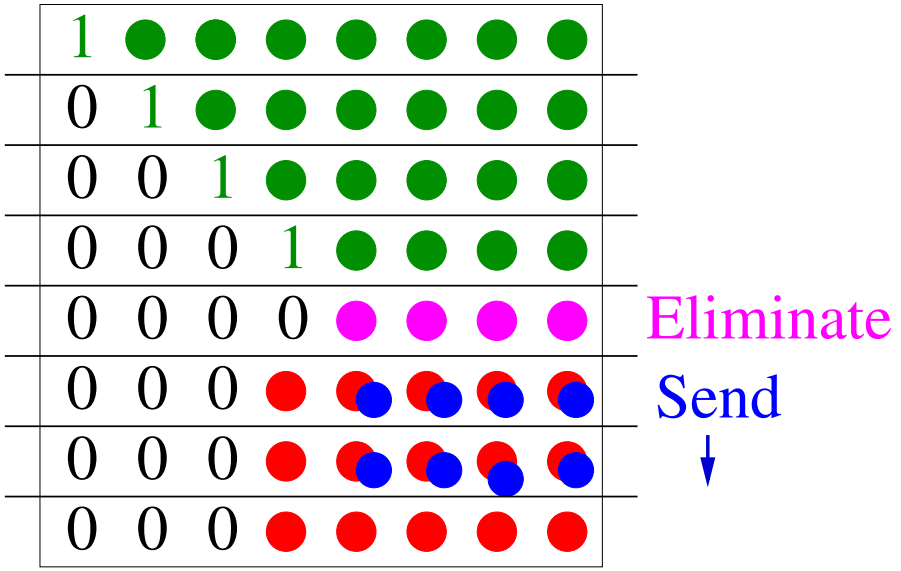
Pipelined Version



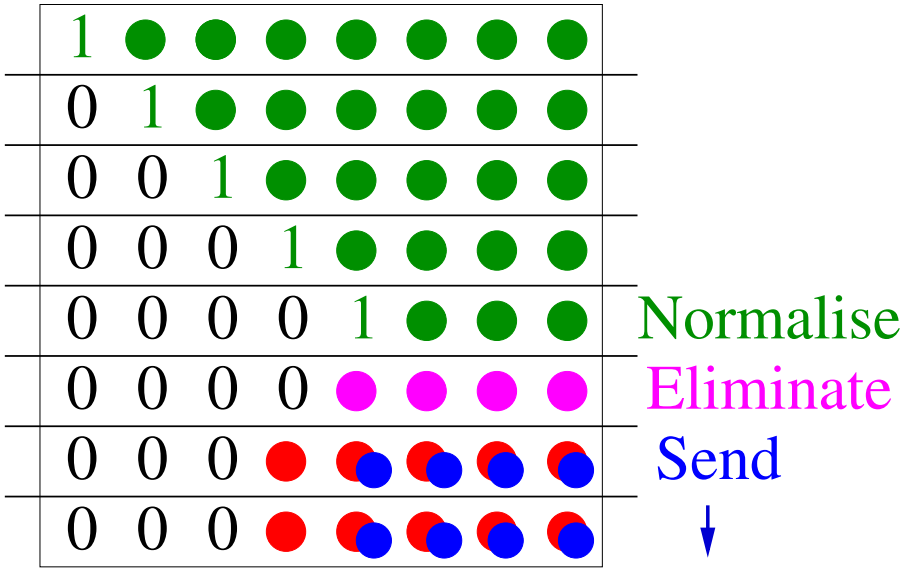
Pipelined Version



Pipelined Version



Pipelined Version



Observations

Observations

Stages have internal state.

Observations

Stages have internal state.

There are no external buffers of input or output (both are in the stage state).

Observations

Stages have internal state.

There are no external buffers of input or output (both are in the stage state).

Sequence of interactions is state dependent (activity is different the "final time").

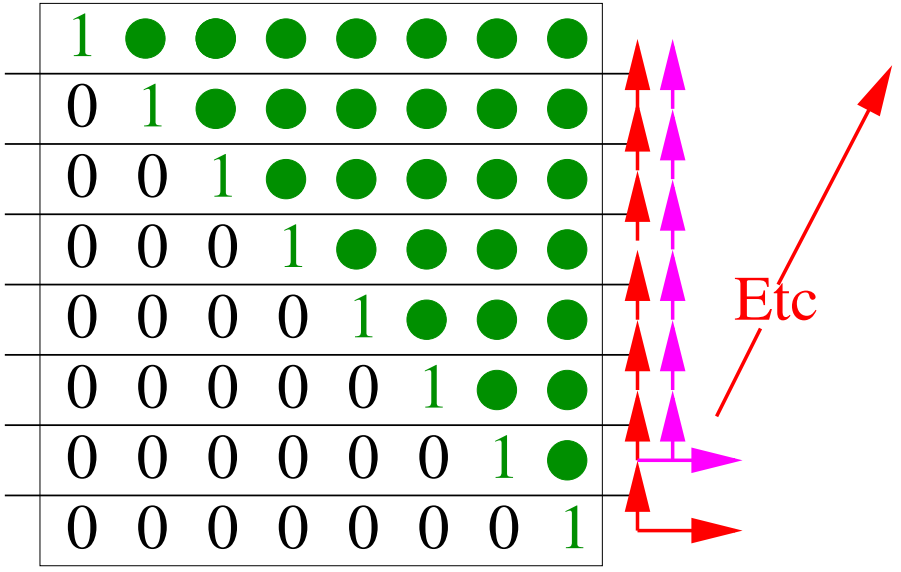
More Pipelining

More Pipelining

A further observation is that the back-substitution phase can be pipelined too, but in the other direction.

More Pipelining

A further observation is that the back-substitution phase can be pipelined too, but in the other direction.



Algorithm Summary

```
Scatter_data();                                          // standard MPI

Pipeline (top-to-bottom, elimination, ....);             // skeleton call

Pipeline (bottom-to-top, back_substitution, ....); // skeleton call

Gather_results();                                        // standard MPI
```

eSkel

eSkel

The edinburgh Skeleton library

eSkel

The edinburgh Skeleton library

An experimental attempt to address these issues.

eSkel

The edinburgh Skeleton library

An experimental attempt to address these issues.

An extension of MPI's collective operation suite.

MPI Key Concepts

MPI Key Concepts

A process (not processor) based model.

MPI Key Concepts

A process (not processor) based model.

Processes identified by ranks within groups (known as "communicators").

MPI Key Concepts

A process (not processor) based model.

Processes identified by ranks within groups (known as "communicators").

Default communicator (all processes) is `MPI_COMM_WORLD`.

MPI Key Concepts

A process (not processor) based model.

Processes identified by ranks within groups (known as "communicators").

Default communicator (all processes) is `MPI_COMM_WORLD`.

Every communication specifies its communicator.

MPI Key Concepts

A process (not processor) based model.

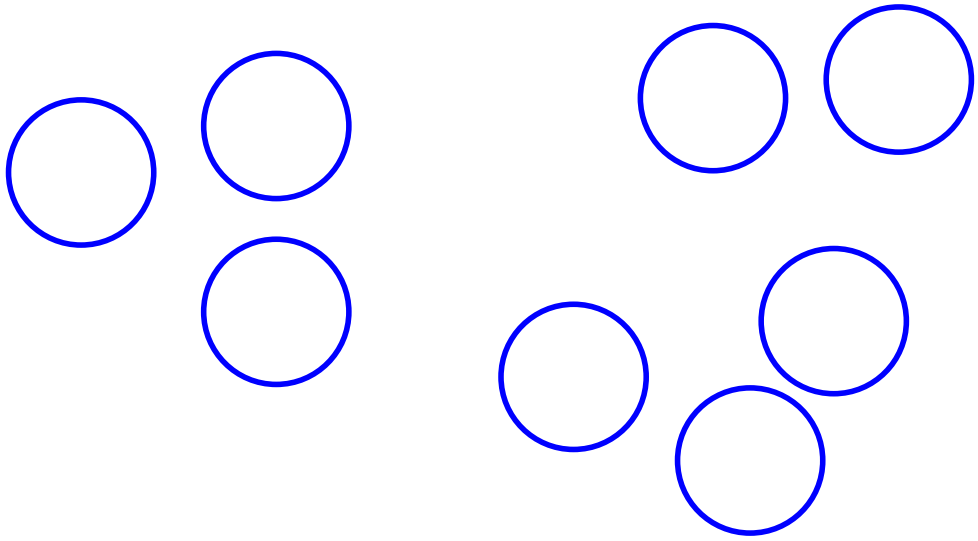
Processes identified by ranks within groups (known as "communicators").

Default communicator (all processes) is `MPI_COMM_WORLD`.

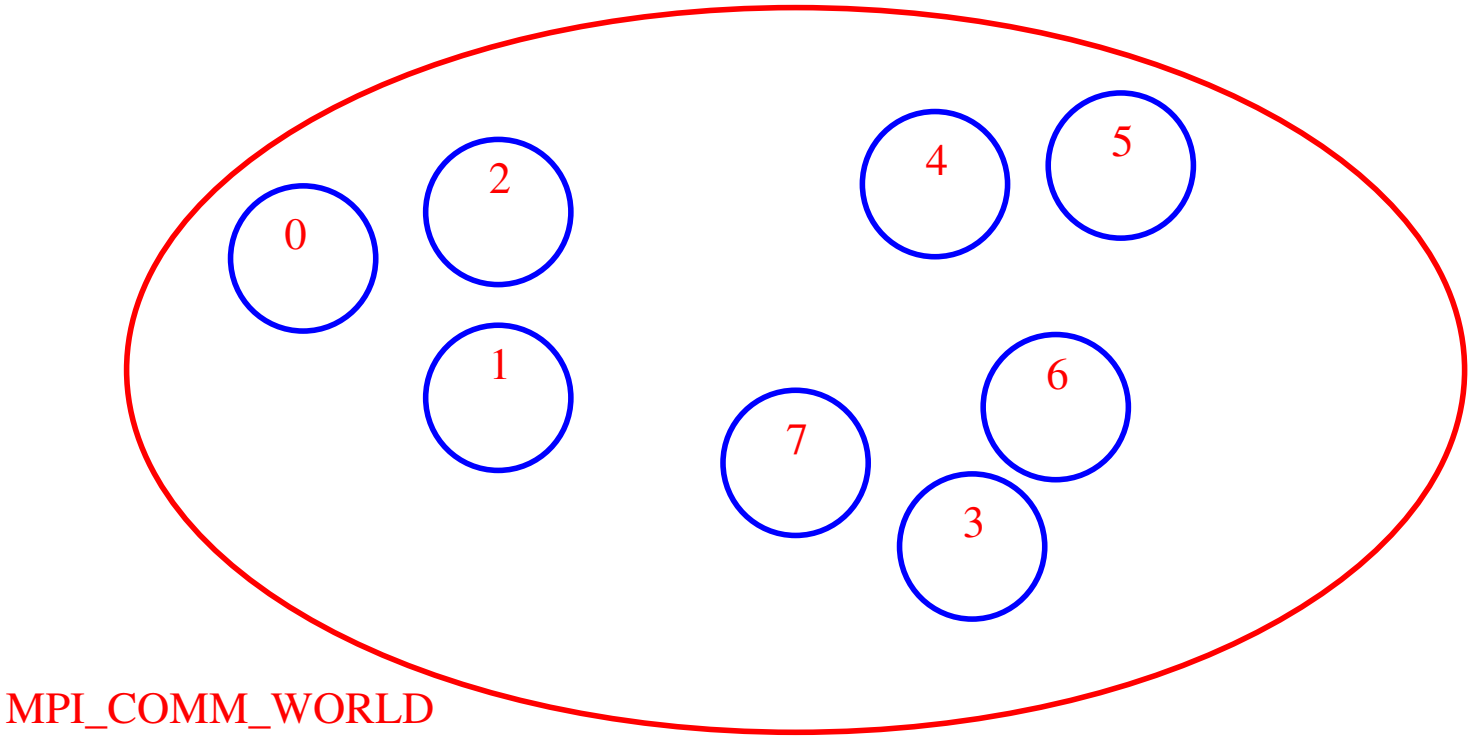
Every communication specifies its communicator.

Allows programmer to reflect logical groupings in an algorithm and to insulate communications within these from "outside" interference.

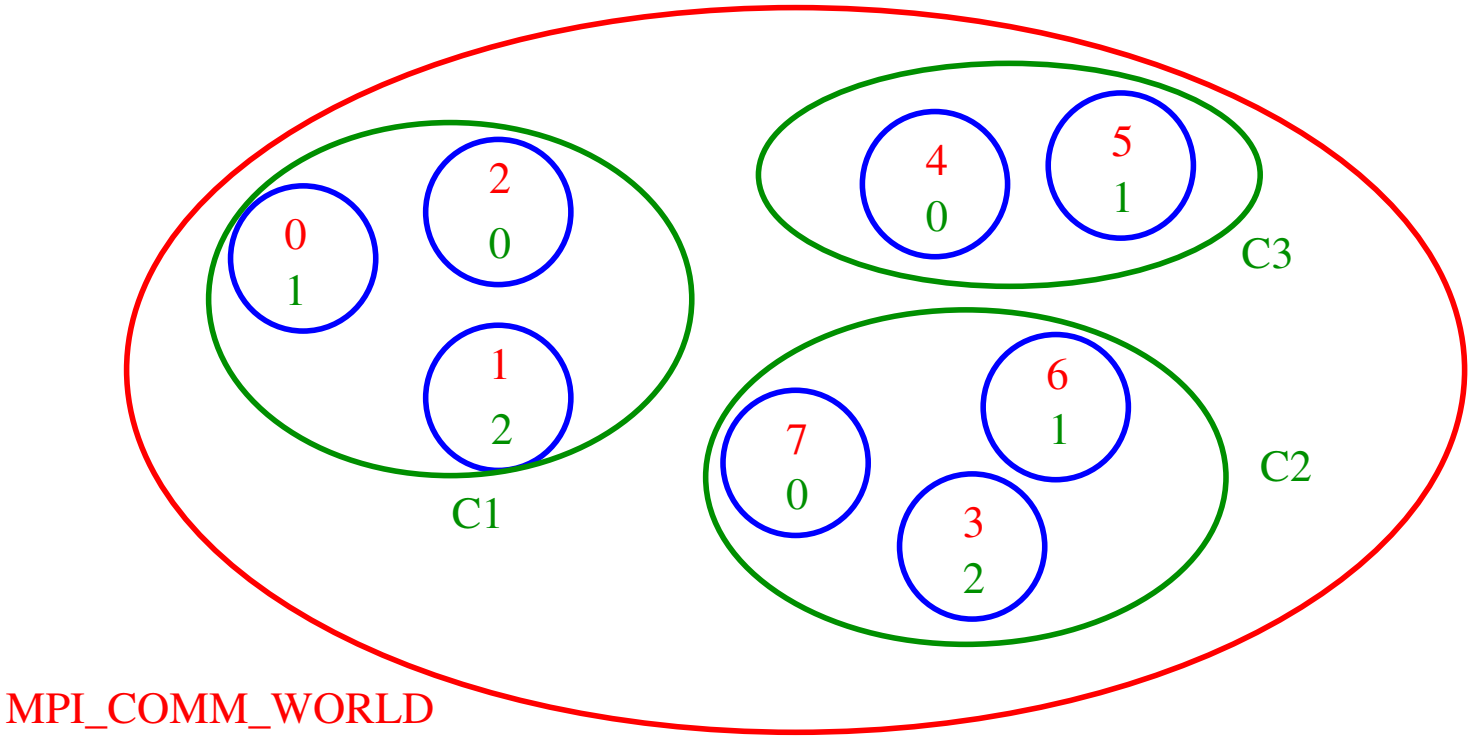
MPI Communicators



MPI Communicators



MPI Communicators



MPI: Collective Operations

```
int MPI_Reduce (void * sndbuf, void * rcvbuf, int count, MPI_Datatype dt,
                MPI_Op op, int root, MPI_Comm comm)
```

- input data buffer (each process contributes)
- output buffer (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)

MPI: Collective Operations

```
int MPI_Reduce (void * sndbuf, void *rcvbuf, int count, MPI_Datatype dt,
                MPI_Op op, int root, MPI_Comm comm)
```

- input data buffer (each process contributes)
- output buffer (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)

MPI: Collective Operations

```
int MPI_Reduce (void * sndbuf, void *rcvbuf, int count, MPI_Datatype dt,
                MPI_Op op, int root, MPI_Comm comm)
```

- input data buffer (each process contributes)
- output buffer (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)

MPI: Collective Operations

```
int MPI_Reduce (void * sndbuf, void * rcvbuf, int count, MPI_Datatype dt,
                MPI_Op op, int root, MPI_Comm comm)
```

- input data buffer (each process contributes)
- output buffer (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)

MPI: Collective Operations

```
int MPI_Reduce (void * sndbuf, void * rcvbuf, int count, MPI_Datatype dt,
                MPI_Op op, int root, MPI_Comm comm)
```

- input data buffer (each process contributes)
- output buffer (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)

eSkel

eSkel

MPI collective operations like `MPI_Reduce` are already simple skeletons.

eSkel

MPI collective operations like `MPI_Reduce` are already simple skeletons.

We design eSkel from this basis to provide

- minimal conceptual disruption (principle 1)
- ad-hoc parallelism (principle 2)

eSkel Skeletons

The current draft of eSkel defines five skeletons

eSkel Skeletons

The current draft of eSkel defines five skeletons

Pipeline

Farm

eSkel Skeletons

The current draft of eSkel defines five skeletons

Pipeline

Farm

Deal

HaloSwap

Butterfly

Deal

Deal

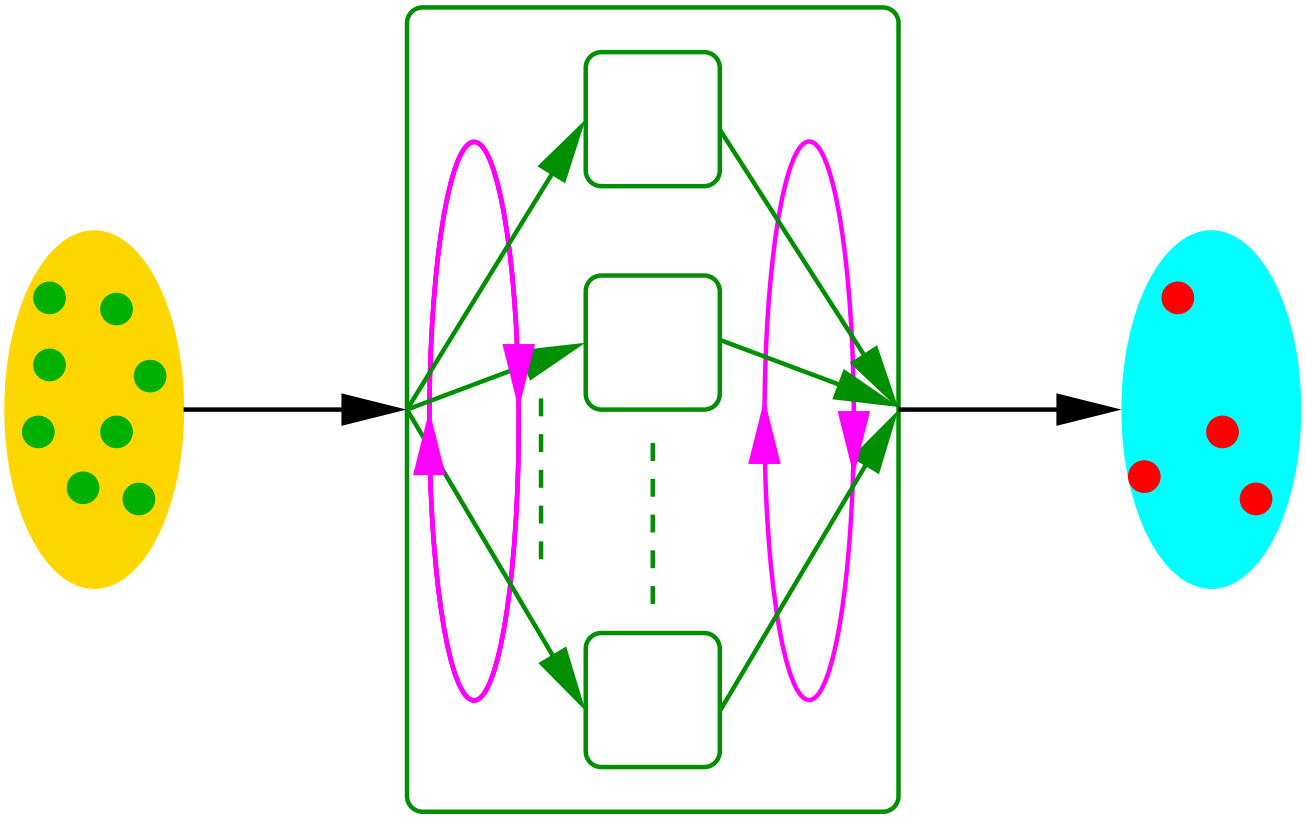
Similar to a farm, but distributes task in cyclic order to workers (no farmer).

Deal

Similar to a farm, but distributes task in cyclic order to workers (no farmer).

Useful nested in pipelines, to internally replicate a stage.

Deal



HaloSwap

HaloSwap

Representative of iterative relaxation algorithms.

HaloSwap

Representative of iterative relaxation algorithms.

Loop over "local update" and "check for termination".

HaloSwap

Representative of iterative relaxation algorithms.

Loop over "local update" and "check for termination".

Interactions have two components (one from each neighbour).

HaloSwap

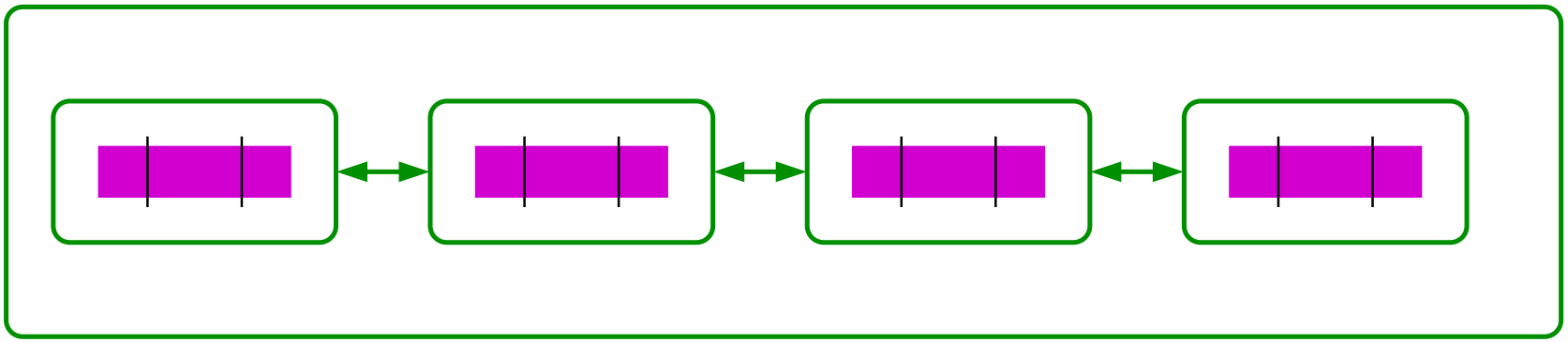
Representative of iterative relaxation algorithms.

Loop over "local update" and "check for termination".

Interactions have two components (one from each neighbour).

Optional wraparound.

HaloSwap



Butterfly

Butterfly

Captures a class of divide-and-conquer algorithms (those based on traversing hypercube dimensions).

Butterfly

Captures a class of divide-and-conquer algorithms (those based on traversing hypercube dimensions).

A sequence of activities, in groups of different sizes.

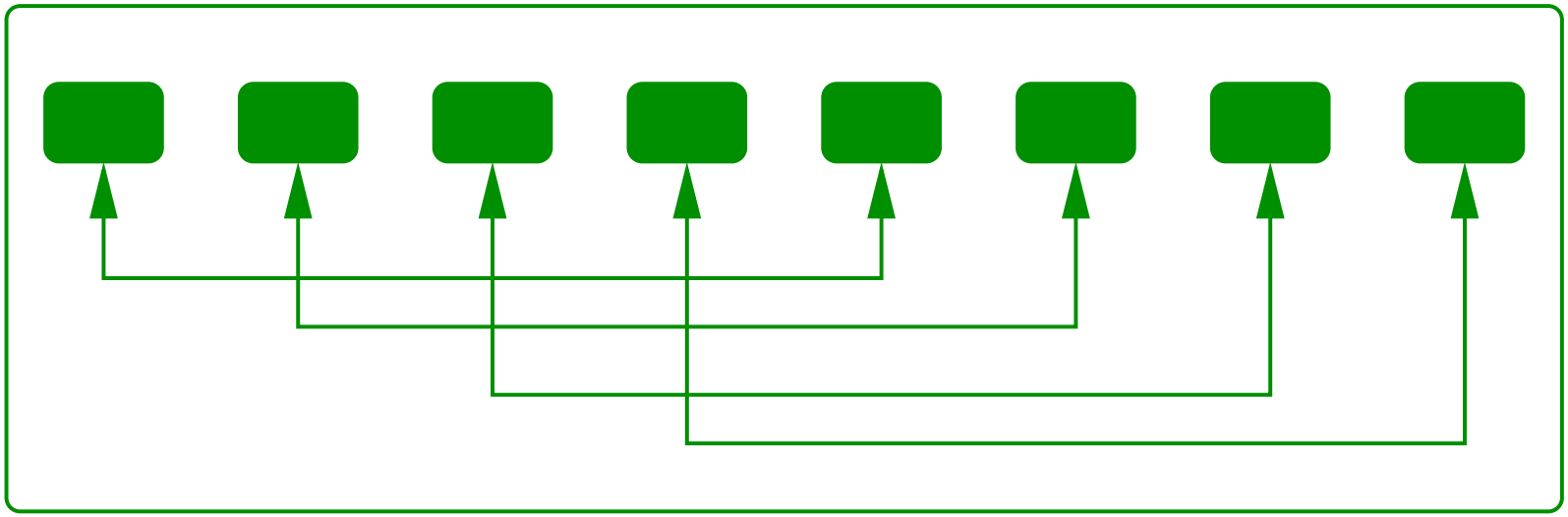
Butterfly

Captures a class of divide-and-conquer algorithms (those based on traversing hypercube dimensions).

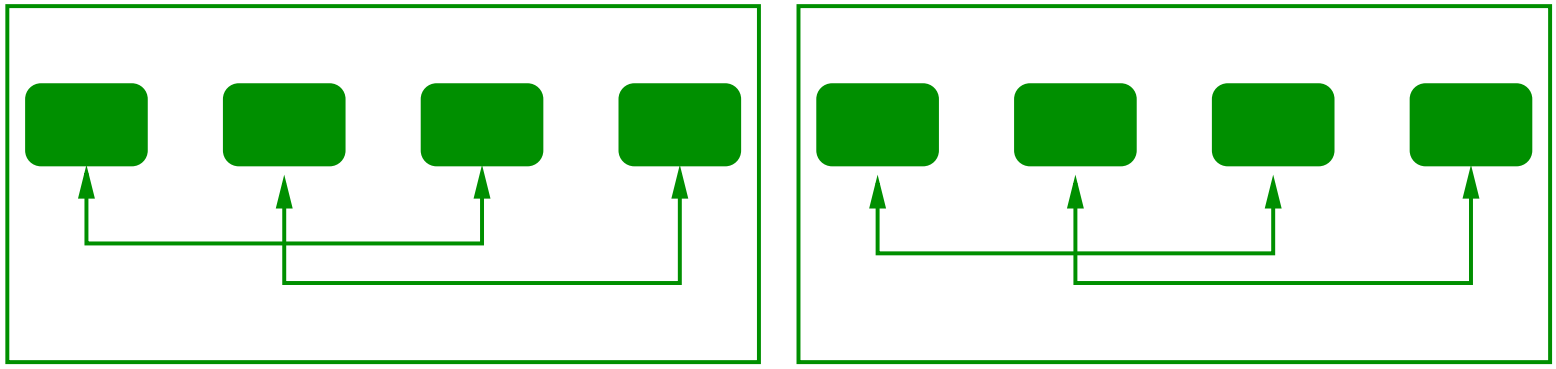
A sequence of activities, in groups of different sizes.

Constrained to work with 2^d processes.

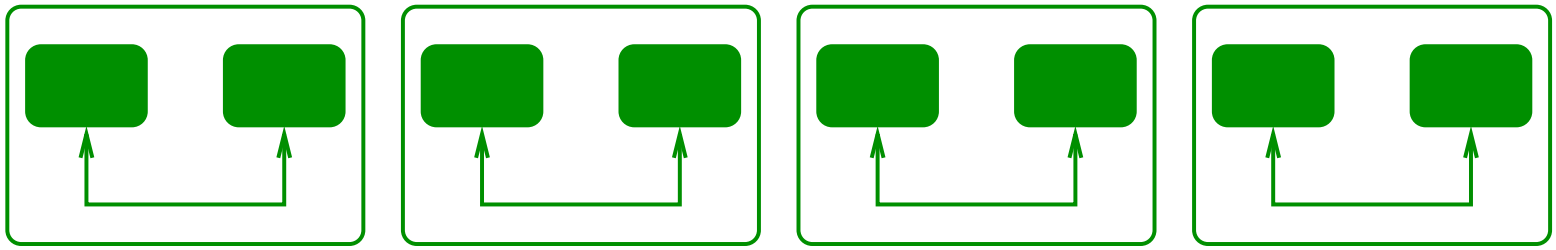
Butterfly



Butterfly



Butterfly



The Gory Details

The Gory Details

Function prototype for the pipeline skeleton:

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

Why do we need fifteen parameters?

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

Why do we need fifteen parameters?

Because of the MPI basis and for flexibility.

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

Why do we need fifteen parameters?

Pipeline inputs

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

Why do we need fifteen parameters?

Pipeline output buffer

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

Why do we need fifteen parameters?

Pipeline stage activities

The Gory Details

Function prototype for the pipeline skeleton:

```
void Pipeline  (int ns, Amode_t amode[],
                eSkel_molecule_t * (*stages[])(eSkel_molecule_t *), int col,
                Dmode_t dmode, spread_t spr[], MPI_Datatype ty[], void *in,
                int inlen, int inmul, void *out, int outlen, int *outmul,
                int outbuffsz, MPI_Comm comm)
```

Why do we need fifteen parameters?

Stage interfaces and modes

Summary

Summary

Parallel programming is important, but hard.

Summary

Parallel programming is important, but hard.

Structured parallel programming can help by allowing the programmer to express meta-knowledge about interaction structure.

Summary

Parallel programming is important, but hard.

Structured parallel programming can help by allowing the programmer to express meta-knowledge about interaction structure.

This information allows the implementation to make macro optimisations, and supports coarse grain algorithm development methodologies.

Summary

Parallel programming is important, but hard.

Structured parallel programming can help by allowing the programmer to express meta-knowledge about interaction structure.

This information allows the implementation to make macro optimisations, and supports coarse grain algorithm development methodologies.

To enter the mainstream we have to be pragmatic.

Future Work

Future Work

These concepts and arguments are generic, and may be applied wherever parallelism appears:

Future Work

These concepts and arguments are generic, and may be applied wherever parallelism appears:

Mainstream parallel computing

Future Work

These concepts and arguments are generic, and may be applied wherever parallelism appears:

Mainstream parallel computing

Grid computing?

Future Work

These concepts and arguments are generic, and may be applied wherever parallelism appears:

Mainstream parallel computing

Grid computing?

ASIC design?

Future Work

These concepts and arguments are generic, and may be applied wherever parallelism appears:

Mainstream parallel computing

Grid computing?

ASIC design?

FPGA programming?

Future Work

Future Work

I would like someone to give me a large amount of money and substantial resources in order to be able to pursue this programme swiftly and comprehensively.

Future Work

I would like someone to give me a large amount of money and substantial resources in order to be able to pursue this programme swiftly and comprehensively.

No reasonable offer refused.

Future Work

I would like someone to give me a large amount of money and substantial resources in order to be able to pursue this programme swiftly and comprehensively.

No reasonable offer refused.

Thank you