Parallel Programming with Skeletons: Past, Present, Future.

Murray Cole

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh
Edinburgh
Why Skeletal Parallel Programming Matters

The skeletal approach to parallelism proposes that commonly used patterns of computation and interaction should be packaged up as parameterisable library functions (or even language control constructs) so that application programmers can benefit from tuned implementations of the underlying structure, without “re-inventing the wheel” and without risking the introduction of complex errors.
Overview

Parallel programming is important, but hard

⇓

Skeletal programming can help

⇓

We have to be pragmatic

⇓

eSkel is a step in this direction
Why Parallel Programming Matters

Parallelism is everywhere, from Google, through SETI type projects, Top 500 “supercomputers”, servers, desk-tops, multi-core chips, to ASICs

Mundane parallelism is spreading ("servers", "clusters", PCs)

As this becomes mainstream, curiosity/greed will get the better of us!
Why Parallel Programming is Hard

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

Expressing (and testing) co-ordination and synchronisation is confusing.

We must do all this and maintain high efficiency in practice (not just big O), otherwise the exercise is pointless.
Why Communication Matters

Parallel programming is about devising and implementing an algorithm in a way which minimises overhead, including extra work and interaction between processes/threads.

Interaction is always about communication

Obvious in message passing models

Also true in shared address space models, where artefactual communication (Culler) of cache lines (or pages) and supporting invalidation traffic, false sharing, etc are very hard to control or even understand.
Why Collective Communication Matters

Gorlatch\textsuperscript{1} argues for the use of collective communication operations over simple send-receive on grounds of

- **Simplicity** (consider broadcast)
- **Programmability** ("SEQ of PAR" methodology)
- **Expressiveness** (list of real examples)
- **Performance** (machine specific)
- **Predictability** (cost modelling seems at least feasible)

\textsuperscript{1}S. Gorlatch, “Send-Recv Considered Harmful? Myths and Truths about Parallel Programming” (PaCT 2001)
Send-Recv Considered Essential?

MPI encoding of some natural programs seem to require Send-Recv

- Pipelined algorithms
- Dynamically load balanced farms

But these both still have conceptual structure (ie unpredictable but not random).

Can we capture this structure and eliminate these explicit send-recvs with an extended collective communication repertoire?
Why Skeletal Parallel Programming Matters

The skeletal approach to parallelism proposes that commonly used patterns of computation and interaction should be packaged up as parameterisable library functions (or even language control constructs) so that application programmers can benefit from tuned implementations of the underlying structure, without “re-inventing the wheel” and without risking the introduction of complex errors.
A Pipeline
A Task Farm
Why Parallel Programming is Hard

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

Expressing (and testing) co-ordination and synchronisation is confusing. Sometimes behaviour is not repeatable.

We must do all this and maintain high efficiency in practice (not just big O), otherwise the exercise is pointless. This is already difficult sequentially (eg blocking for cache performance).
Does Skeletal Programming Help?

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

Expressing (and testing) co-ordination and synchronisation is confusing. Sometimes behaviour is not repeatable.

We must do all this and maintain high efficiency in practice (not just big O), otherwise the exercise is pointless. This is already difficult sequentially (eg blocking for cache performance).
Does Skeletal Programming Help?

No, this is still hard! Skeletons help us express parallel algorithms but we still have to devise the algorithm.

Expressing (and testing) co-ordination and synchronisation is confusing. Sometimes behaviour is not repeatable.

We must do all this and maintain high efficiency in practice (not just big O), otherwise the exercise is pointless. This is already difficult sequentially (eg blocking for cache performance).
Does Skeletal Programming Help?

No, this is still hard! Skeletons help us express parallel algorithms but we still have to devise the algorithm.

Yes! The pattern abstracted by the skeleton can hide all the details of interaction, which are common to all its uses.

We must do all this and maintain high efficiency in practice (not just big O), otherwise the exercise is pointless. This is already difficult sequentially (eg blocking for cache performance).
Does Skeletal Programming Help?

No, this is still hard! Skeletons help us express parallel algorithms but we still have to devise the algorithm.

Yes! The pattern abstracted by the skeleton can hide all the details of interaction, which are common to all its uses.

Yes! Like any library, the skeleton implementation can use the best-known techniques to extract performance. It knows the interaction pattern in advance, and can exploit this knowledge at a low level.
A Pipeline - Image Processing

Decreasing Data Size

Edges → Faces → Objects → Scene

Increasing Content
Observations

Each input produces exactly one output

Stages may have internal state (e.g. producing a report on the scene may depend upon what has been seen previously - “the car is turning left”)

Stages may need internal parallelism to balance throughput (e.g. pixel processing)
A Pipeline - Gaussian Elimination
Elimination Phase

\[
\begin{array}{cccccccc}
1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 1 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & 1 & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & 0 & 1 & \bullet & \bullet & \bullet \\
0 & 0 & 0 & 0 & 0 & 1 & \bullet & \bullet \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]
Back Substitution Phase
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

\[
\begin{array}{ccccccc}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

Pivot Row
Elimination Phase

\[
\begin{array}{cccccccc}
1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 1 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & 1 & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Normalise
Elimination Phase

```
1
0 1
0 0 1
0 0 0 1
0 0 0
0 0 0
0 0 0
```

Broadcast
Elimination Phase

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Eliminate
Eliminate
Eliminate
Eliminate
Eliminate
Eliminate
Eliminate
Eliminate
Elimination Phase

```
1  •  •  •  •  •  •  •
0 1  •  •  •  •  •  •
0 0 1  •  •  •  •  •
0 0 0 1  •  •  •  •
0 0 0 0 •  •  •  •  •
0 0 0 0 •  •  •  •  •
0 0 0 0 •  •  •  •  •
0 0 0 0 •  •  •  •  •
```

Pivot Row
Pipelined Version

Improved algorithm (see Kumar) interleaving broadcast and elimination.

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

\[
\begin{array}{cccccccc}
1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 1 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 1 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & 1 & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Normalise
Pipelined Version

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

Send

\[\rightarrow\]
Pipelined Version

\[
\begin{array}{cccccccccccccc}
1 & & & & & & & & & & & & \\
0 & 1 & & & & & & & & & & & \\
0 & 0 & 1 & & & & & & & & & & \\
0 & 0 & 0 & 1 & & & & & & & & & \\
0 & 0 & 0 & & & & & & & & & & \\
0 & 0 & 0 & & & & & & & & & & \\
0 & 0 & 0 & & & & & & & & & & \\
0 & 0 & 0 & & & & & & & & & & \\
\end{array}
\]

Send
Pipelined Version

\[\begin{array}{c}
1 \\
0 1 \\
0 0 1 \\
0 0 0 1 \\
\end{array}\]

Eliminate

Send

\[\begin{array}{c}
0 0 0 0 \\
0 0 0 \\
0 0 0 \\
0 0 0 \\
\end{array}\]
Pipelined Version

\[
\begin{array}{c}
1 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
\end{array}
\]

Normalise
Eliminate
Send

Send
Eliminate
Normalise
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”)

Murray Cole
Parallel Programming with Skeletons
November 2003
Pipelining Again

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

![Diagram showing pipelining process]

Etc
Program Summary

Scatter_data(); // standard MPI

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

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

Gather_results(); // standard MPI
Skeleton Programming Projects
Skeleton Programming Projects
Skeleton Programming Projects

- llc (right here!)
- Eden (Madrid/Marburg)
Skeleton Programming Projects

Too many others to list, but many approaches

- pure **functional programming** (skeletons as higher order functions)
- **imperative** (skeletons as libraries/control constructs)
- **mixed** functional/imperative (as above, controlling imperative activities)
- **pattern/object oriented** (with “open” templates)
Who cares?

Skeletal programming remains a fringe activity

But,

- **MPI** has (some) similar aims and is dominant

- **Patterns** have a similar basis and are in widespread use
A Pragmatic Manifesto

1. Introduce with minimal conceptual disruption

- skeletal programming is not necessarily functional

- skeletal programming is not necessarily OO

- parallel programmers are happy with C/Fortran and MPI!

- we should build on this, not dismiss it
A Pragmatic Manifesto

2. Integrate ad-hoc parallelism

• some parallelism seems inherently unstructured
  – I/O & data distribution
  – non-determinism

• allow contained integration

• provide a toolkit, not a straitjacket
A Pragmatic Manifesto

3. Accommodate diversity

- “Folklore” concepts tend to diverge under tight specification
  - Pipelines
    * stage as function v. stage as process
    * “one-for-one” v. arbitrary interaction
  - Farms
    * implicit v. explicit farmer
    * “one-for-one” v. arbitrary

- provide a toolkit, not a straitjacket
A Pragmatic Manifesto

4. *Show the pay-back*

- Real benefits with small effort

- Substantial case studies
eSkel

The edinburgh Skeleton library

A prototype of a draft of one response!

An extension of MPI’s collective operation suite
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

- powerful operations to create new communicators from existing ones (duplicate, partition, etc)

- 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_COMM_WORLD

0

1

2

3

4

5

6

7
MPI: Simple Operations

```c
int MPI_Send (void * buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm)
```

Specify

- data as a triple (pointer, type, count)
- destination as a rank within a communicator
- extra context as a “tag”
MPI: Simple Operations

int MPI_Send (void * buf, int count, MPI_Datatype dt, int dest, int tag, MPI_Comm comm)

Specify

• data as a triple (pointer, type, count)

• destination as a rank within a communicator

• extra context as a “tag”
MPI: Simple Operations

int MPI_Send (void * buf, int count, MPI_Datatype dt, int dest, int tag, MPI_Comm comm)

Specify

• data as a triple (pointer, type, count)

• destination as a rank within a communicator

• extra context as a “tag”
MPI: Simple Operations

```
int MPI_Send (void * buf, int count, MPI_Datatype dt, 
int dest, int tag, MPI_Comm comm)
```

Specify

- data as a triple (pointer, type, count)
- destination as a rank within a communicator
- extra context as a “tag”
MPI: Collective Operations

All processes in comm must call (otherwise deadlock).

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

- input data buffer as before (but now each process contributes)
- output buffer similar (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)
MPI: Collective Operations

All processes in comm must call (otherwise deadlock).

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

- input data buffer as before (but now each process contributes)
- output buffer similar (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)
MPI: Collective Operations

All processes in comm must call (otherwise deadlock).

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

- input data buffer as before (but now each process contributes)
- output buffer similar (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)
MPI: Collective Operations

All processes in comm must call (otherwise deadlock).

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

- input data buffer as before (but now each process contributes)
- output buffer similar (note restriction on type)
- operation to be used
- group and special roles within it (in this case, root receives the result)
MPI: Collective Operations

All processes in comm must call (otherwise deadlock).

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

- input data buffer as before (but now each process contributes)

- output buffer similar (note restriction on type)

- operation to be used

- group and special roles within it (in this case, root receives the result)
**eSkel**

- MPI collective operations like `MPI_Reduce` are already like simple skeletons!

- We design the eSkel library around this basis, therefore immediately addressing issues of
  - **minimal conceptual disruption** (principle 1)
  - **ad-hoc parallelism** (principle 2)

- **accommodating diversity** (principle 3) requires careful thought

- **showing the pay back** (principle 4) requires time and effort
eSkel Skeletons

eSkel has a pipeline skeleton:

```c
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 outbuffersz, MPI_Comm comm)
```

Why do we need fourteen parameters?!
eSkel Skeletons

eSkel has a pipeline skeleton:

```c
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 fourteen parameters?!

Because of the MPI basis and some new concepts.
Data in eSkel

Consider the input data involved in a pipeline, MPI style.
Data multiplicity in eSkel

- each process may want to provide some inputs (as in MPI_Reduce)

- each such item may be described by a (pointer, type, count) triple

- each process may provide a different number of items
  - in eSkel, we call this multiplicity, to distinguish it from the length of a particular item
  - (contrast with MPI which mixes these concepts, arguably inconsistently)

Each calling process must specify the multiplicity of its inputs, and provide a parameter through which it can be told the multiplicity of its outputs.
Data spread in eSkel

In MPI applications, it is common to partition and distribute a large data structure (such as an image) across a set of processes.

In MPI the programmer has no way of indicating that this has happened!

Suppose eSkel processes want to provide such data as input to skeletons.

We use the new concept of spread to explicitly state whether the items provided form part of larger whole ("global spread") or are complete in their own right.
Data spread in eSkel

Local Spread: 12 distinct items

OR

Global Spread: 4 distinct items
eSkel Data Parameters

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)

• input data

• output data and buffer

• interactions between stages
eSkel Data Parameters

```c
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)
```

- input data
- output data and buffer
- interactions between stages
eSkel Data Parameters

```c
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)
```

- input data
- output data and buffer
- stage interfaces
eSkel Activities

In general, a skeleton co-ordinates the actions of sub-groups of its processes (e.g., pipeline stages)

- eSkel calls these sub-groups activities and represents them with MPI communicators

- activities may be internally parallel (either ad-hoc, or with nested skeletal structure)

- some skeletons may specify dynamic activity structure (not pipelines, but consider typical divide & conquer)
Specifying Activities

How do we specify activities in a generic way?

- might like to say that an activity is a $\alpha \Rightarrow \beta$ function, where $\alpha$ and $\beta$ are tied to the data specifications

- we can’t say this in C

- introduce eSkel_molecule_t as a generic means of capturing an activity’s inputs (and outputs)

- essentially a wrapper for some void * pointers, lengths and types

- activities take a molecule as parameter and return a molecule as a result
Activity Parameters

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)

• a communicator (in the style of MPI collectives)

• the number of stages

• activity code for the stages

• allocation of processes to stages
Activity Parameters

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)

• a communicator (in the style of MPI collectives)

• the number of stages

• activity code for the stages

• allocation of processes to stages
Activity Parameters

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)

• a communicator (in the style of MPI collectives)

• the number of stages

• activity code for the stages

• allocation of processes to stages
Activity Parameters

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)

- a communicator (in the style of MPI collectives)
- the number of stages
- activity code for the stages
- allocation of processes to stages
Activity Modes

“might like to say that an activity is a $\alpha \Rightarrow \beta$ function, where $\alpha$ and $\beta$ are tied to the data specifications”

But some activities don’t behave like functions!

Consider

• a pipeline stage which behaves like a filter

• a farm worker which generates several results for each task

Activities like this need more control over their interactions.
Generic Interactions: Give & Take

- Generic operations to
  - Give (the next molecule to the skeleton)
  - Take (the next molecule from the skeleton)

- API is generic, semantics are skeleton specific
  - in a Farm a worker activity calling Take receives a new task molecule from the farmer without knowing anything about where the farmer is or how it is implemented
  - in a Pipeline a stage activity calling Give causes a new result molecule to be delivered without knowing anything about where to, or even whether the destination is an activity or a data collection
Activity Modes

Activities calling Give and Take don’t use the parameter/return mechanism.

They should “really” be typed `void Activity (void)`

Instead, we introduce the concept of activity mode, to indicate an activity’s status

- **implicit mode**: library code handles interactions and calls activity code *transiently* as required

- **explicit mode**: activities call their own interactions and so are called once *persistently* for the whole skeleton call

- **skeleton mode**: indicates that the activity will call another skeleton (nested) whose activity mode will control behaviour.
Activity Modes

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)

- activity modes for each stage
Data Modes

On a coarse scale, a skeleton call is simply a function which consumes a data collection and produces a data collection.

These collections can either be provided as parameters to the skeleton call (MPI style), or streamed into the skeleton from an enclosing skeleton call.

Programmer uses eSkel’s data mode to specify the choice.

- buffered mode
- streamed mode
Pipeline (.......Buffered Mode .......)
Pipeline (.......Buffered Mode .......)
Pipeline (.......Buffered Mode .......)  

Deal (.......Stream Mode .......)
Pipeline (.......Buffered Mode .......) 

Deal (.......Stream Mode .......)
Data Modes

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)

- data mode for the whole pipeline
eSkel Skeletons

The current draft of eSkel defines five skeletons

- Pipeline
- Farm
- Deal
- HaloSwap
- Butterfly

This is definitely not a complete set, and probably not even the ultimate definition of those present.
Deal

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

Useful nested in pipelines, to internally replicate a stage

Natural data mode is buffered

Only implicit or skeleton activity mode
HaloSwap
HaloSwap

Loop over two activities, “local update” and “check for termination”

Molecules have two components (one from each neighbour)

New generic function Exchange (in explicit activity mode) swaps the “halo”. Give and Take interact with data buffers or streams.

Optional wraparound
Butterfly
Butterfly
Butterfly
Butterfly

Captures some divide-and-conquer algorithms (those based on traversing hypercube dimensions)

A sequence of activity activations, in groups of different sizes.

Constrained to work with $2^d$ processes

Exchange (in explicit activity mode) interacts with the partner in the current hypercube dimension. Give and Take interact with data buffers or streams.
Mandelbrot as a Farm

Create distributed array of points (independent SPMD)

Farm the calculations on chunks of points (skeleton)

Gather and output results (straight MPI)

Sequential per pixel code lifted verbatim from the web
typedef struct { int r; int g; int b; } pixel_t;
typedef struct { int x; int y; } point_t;
point_t point[x*y]; pixel_t pixel[x*y];

MPI_Type_contiguous (2, MPI_INT, &MPI_POINT);
MPI_Type_contiguous (3, MPI_INT, &MPI_PIXEL);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &me);

Farm (p, IMPL, mandelcheck, me
   (void *) point, CHSIZE, imul, SPLOCAL, MPI_POINT,
   (void *) pixel, CHSIZE, &omul, SPLOCAL, MPI_PIXEL,
   x*y, mycomm() );
The Worker Activity

eSkel_molecule_t *mandelcheck (eSkel_molecule_t *p) {
    point_t *points;  pixel_t *pix;
eSkel_molecule_t *result;
    points = (point_t *) (p->data[0]);
    pix = (pixel_t *) malloc (sizeof (pixel_t)*CHUNK);
    for (i=0; i<CHSIZE; i++) {
        // original code computing pix[i] from points[i]
    }
    result = (eSkel_molecule_t *) malloc (sizeof (eSkel_molecule_2));
    result->data[0] = (void *) pix;
    result->len[0] = CHSIZE;
    return result;
}
Performance

Earlier version of eSkel API, dedicated farmer process. 24 processor SunFire SMP.
Parallel programming is important, but hard

Skeletal programming can help

We have to be pragmatic

eSkel is a step in this direction