Pluggable Parallelisation

Rui C. Gonçalves, João L. Sobral

High Performance Distributed Computing
Munich, 11 June 2009

Outline

- Motivation
- The approach
- Pluggable parallelisation
- Composition of abstractions
- Evaluation
  - Parallelisation of legacy code
  - Performance
  - Productivity
- Implementation overview
- Conclusion

Tangled Code (Mix of concerns)

```
public class JGFLUFactBench extends Linpack implements JGFSection2{

  public static int nprocess;
  public static int rank;

  public void JGFinitialise() throws MPIException{
    int r_count,z_count;
    int p_ldaa;
    n = datasizes[size];
    ipvt = new int [ldaa];
    p_ldaa = (ldaa + nprocess - 1) / nprocess;
    rem_p_ldaa = (p_ldaa*nprocess) - ldaa;
    /* ... */

    if(rank==0) {
      long nl = (long) n; //avoid integer overflow
      ops = (2.0*(nl*nl*nl))/3.0 + 2.0*(nl*nl);
      norma = matgen(a,lda,n,b);
    }

    if(rank==0) {
      for(int i=0;i<a.length;i++){
       if(r_count==0) {
         for(int l=0;l<a[0].length;l++){
           buf_a[z_count][l] = a[i][l];
         }
         z_count++;
       } else {
         MPI.Send(a,i,1,MPI.OBJECT,r_count,10);
       }
       buf_list[i] = z_count - 1;
      }
    } else {  // rank!=0
      for(int i=0;i<real_p_ldaa;i++){
        MPI.Recv(buf_a,i,1,MPI.OBJECT,0,10);
      }
    }
  }
```

2

1

3

3

- Basic functionality
- MPI control related issues
- Data partition issues

1. Tasks executed by master process
2. Data partition code
3. Process exchange messages to send/receive data

Motivation (1)

- How to address the complexity of the development of parallel/Grid applications?
- Pluggable parallelisation: separated, incremental, (un)pluggable
  - Better separation of parallelisation concerns
    - Encapsulate each "concern" into a separate abstraction
  - Incremental and (un)pluggable development
    - Start with a "sequential like" application
    - Improve parallelisation in small steps:
      - Correctness can be verified incrementally
  - Support for parallelisation of "legacy code" with less changes than competitive approaches
    - Promote non-invasive parallelisation of "sequential-like" codes

Motivation (2)

- (Expected) benefits of separated, incremental and (un)pluggable parallelisation
  - Easier to develop/maintain/change the parallelisation
  - Increase code reuse across platforms
  - Independent development
  - Better composability of parallelisation
  - Extensibility of abstractions

Approach (1)

- Start with a *sequential-like* code
  - Specify domain specific computations
  - Code amenable for parallelisation

- Pluggable parallelisation specialises (i.e., refines) *sequential-like* code for target platforms
  - OpenMP- and MPI-*equivalent* parallel programming abstractions
  - Can be composed (function composition)
  - Case specific specialisations

Approach (2)

- Supported transformations (i.e., refinements)
  - Control view
    - Block **B** of sequential code executes on all nodes;
    - Block **B** of code executes on nodes N where condition C holds;
  - Data view
    - Data structure **D** is partitioned among nodes using strategy S;
    - Update remote data at execution point **E**.

- Transformations act on program elements (in object-oriented languages)
  - **B**lock of code (method)
  - **D**ata structure (object field)
  - **E**xecution point (method call)

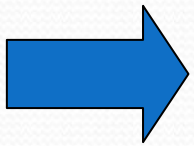
Approach (3)

- Underlying concept: class refinement
  - Adds fields, methods, or constructors
  - Extends or overrides methods and constructors
  - Maintains the name of the class
    - Key feature for scalable composition
- Example: execution of a method on a new thread

```
class SomeClass {
   void Do() { … }
}
```

+

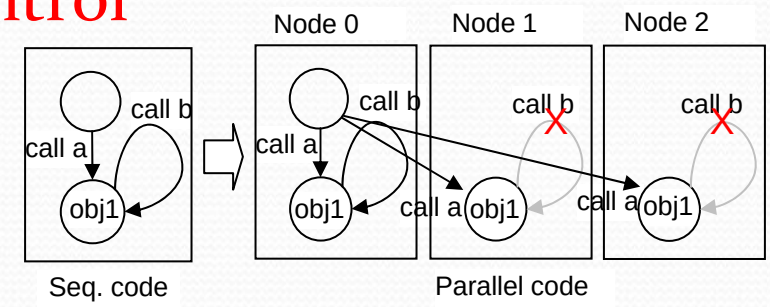
```
OneWay<SomeClass,Do>
```



```
class SomeClass {
   void Do() {
      new Thread() {
         void run() {
            …   // old Do definition
         }
      }
   …
```

Pluggable Parallelisation to …

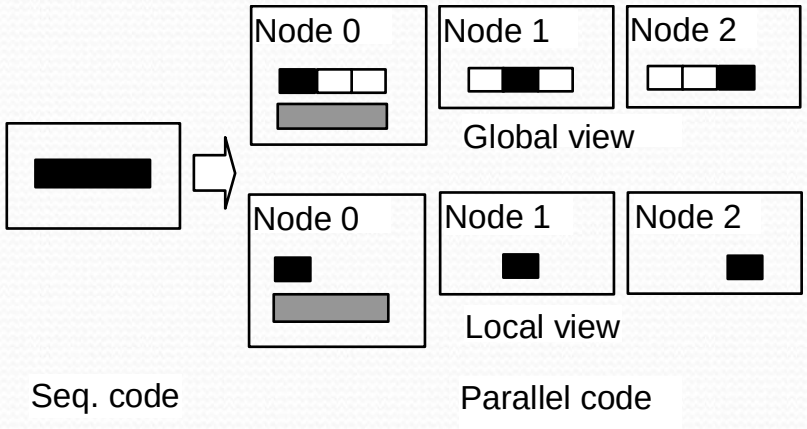
- Transform the sequential control flow into parallel flows of control



Transformation	Description
Replicate<T> (e,g., Replicate<obj1>)	Create an instance of class T on each available resource (e.g. node)
Broadcast<T,M> (see call a)	Execute method call M on all aggregate elements (e.g., nodes)
CondExe<T,M,C> (see call b)	Restrict execution of method M to places where condition C holds

Pluggable Parallelisation to …

- Data partitioning and update



Transformation	Description
Partitioned<T,D, [BLOCK,CYCLE]>	Distribute the data field D, of class T, using a partition strategy
ChangeView<T,E,D, [Local|Global]>	Change field D from/to local/global view at execution point E
Scatter*/Gather* <T,E,D>or<T,E,M>	Update copy of field D or method parameter M at execution point E

Example (JGF Series)

Core functionality

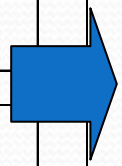
```
class SomeClass {
  double[] TestArray = ... // initialise array
  void Do() {
    for (int i = 0; i < TestArray.length; i++)
      TestArray[i] = someComputation(/*.. */);
  }
}
```

Parallelisation (data view)

```
Partitioned<SomeClass,TestArray,BLOCK>
ScatterBefore<SomeClass,Do,TestArray>
GatherAfter<SomeClass,Do,TestArray>
```

Parallelisation (control view)

```
Replicate<SomeClass>
// all aggregate elements execute method Do
Broadcast<SomeClass,Do>
```



```
class SomeClass {
...
  // Partitioned<SomeClass,TestArray,BLOCK>
  double TestArray[] = new ...
...
  void Do() {
    // ScatterBefore<SomeClass,Do(),TestArray>
    // ChangeView<SomeClass,Do(),TestArray,LOCAL>
    ...
    for (int i = 0; i < TestArray.length; i++)
      TestArray[i] = someComputation(/*.. */);
    }
    // GatherAfter<SomeClass,Do(),TestArray>
    // ChangeView<SomeClass,Do(),TestArray,GLOBAL>
  }
}
```

Broadcast<SomeClass,Do> - Method *Do* executes on all nodes

Partitioned<SomeClass,TestArray,BLOCK> - Partition TestArray block-wise among nodes

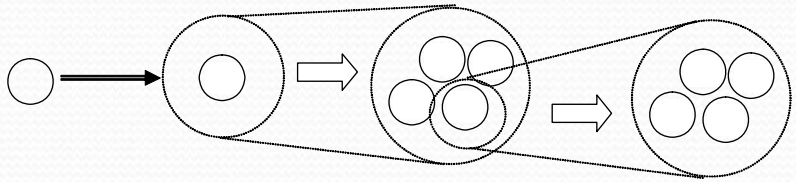
ScatterBefore<SomeClass,Do,TestArray> - Update partitions on remote nodes

ChangeView<SomeClass,Do(),TestArray,LOCAL> - TestArray referes to the local partition

…

Composability & Extensability

- Apply multiple abstractions to a class (or set of classes)
  - Specified by template nesting
    - Similar to an inheritance chain where all sub-classes "collapse" into the base class
    - E.g. Aggregate of an object aggregate: Replicate<Replicate<T>>



- Develop new templates
  - Case specific (see implementation)
  - By composing templates

```
Farm<Class T, Method compute, DataField field>{
    prog1= Replicate<T>
    prog2 = Broadcast<prog1, T, compute>
    prog3 = ScatterBefore<prog2, T, compute, field>
    prog4 = GatherAfter<prog3, T, compute, field>
}
```

Applicability

- Data partition can be changed/refined independently
  - E.g., change from Block to Cyclic etc.
  - Case specific data partitions
    - JGF Crypt: requires data blocks multiples of 8 in
    - SOR performing several iterations without communication
- Shared / distributed memory versions sharing refinements
  - Shared memory may not require data partitions
- Hybrid MPI/OpenMP parallelisation
  - Separately develop SM and DM parallelisation and apply both
    - Targets computational Grids (clusters of multi-core machines)
- Reuse of compositions
  - Identical set of refinements for ASP and LUFact

Evaluation (1)

Abstractions utilization in the JGF benchmarks (DM)

	Allocate	ScatterBefore	GatherAfter	ChangeViewBefore	ChangeViewAfter	ReduceAfter	AllReduceAfter	CondExe	CondExeBCast	CaseSpecific
Crypt	3	1	1	2	1			2		
Series	1		1					3		
SOR	1	1	1					3		1
LUFact	1	1	1					3	4	
RayTracer	1		1	1		1		2		
MonteCarlo	2	1	1	1		1		2		
SparseMatmult	3	3			3	1		2		
MD							6	2		

Evaluation (2)

Speedups (distributed memory) - (shared memory)

Application	4 cores		16 cores		32 cores	
	HW	PP	HW	PP	HW	PP
CryptC	3.54	3.53	7.80	7.56	9.56	9.27
SeriesC	3.11	3.13	12.26	12.29	24.42	24.61
SparseMatmultC	2.11	2.12	6.85	8.28	10.53	19.46
LUFactC	2.22	2.29	2.85	3.03	2.07	2.53
SORC	1.53	1.25	2.93	1.86	2.87	2.49
MDB	3.78	3.74	9.94	10.89	-	-
RayTracerB	3.82	3.88	14.13	14.38	25.64	26.16

8 bi-Xeon 5130 machines
(2x2 cores per machine)

Application	4 cores		8 cores	
	HW	PP	HW	PP
CryptC	3.7	4.1	7.0	7.5
SeriesC	3.4	3.7	7.9	7.9
SparseMatmultC	4.1	4.3	8.3	3.0
LUFactC	3.6	3.6	5.7	5.5
SORC	3.7	3.9	5.9	6.7
MDB	2.9	2.4	4.2	3.2
RayTracerB	3.3	3.8	7.5	7.2

bi-Xeon E5430
(2x4 cores)

Legacy Code (1) - Refactorings

- Pluggable parallelisation may require changes to the code
  - Expose joinpoint – Create a new (named) place to "plug" // code
  - Expose context – Expose domain-specific information in a way to allow the pluggable access the necessary information

- We classify re-factorings in:
  - Expose joinpoint
    - **Move to Method (M2M)** - move a code block to a new method
    - **Move Method Call (MMC)** - move a code block from a place to another
  - Expose context
    - **Move to Object Field (M2OF)** - move a variable to an object field
    - **Processing Dependent of Parameter (PDP)** - change a method to depend on a parameter

Legacy Code (2) - Refactorings

	Expose join point	Expose context
Crypt	2xM2M (G)	-
Series	2xM2M (B)	-
SOR	M2M (G), MMC (G)	M2OF (G)
LUFact	3xM2M (2G/B)	-
RayTracer	M2M (G)	2xM2OF (G/B), PDP (G)
MonteCarlo	M2M (G)	PDP (N)
SparseMatmult	M2M (G), MMC (N)	-
MD	M2M (G)	-

Type of Refactoring

- Move to Method (M2M)
- Move Method Call (MMC)
- Move to Object Field (M2OF)
- Processing Dependent of Parameter (PDP)

Impact of the Refactoring

- Good (G)
- Neutral (N)
- Bad (B)

Productivity assessment

- Number of non commenting source statements measured with the JavaNCSS tool

Application	Base code	JOMP		MPI Java		PP	
	NCSS	NCSS	Grow	NCSS	Grow	NCSS	Grow
Crypt	190	193	2%	242	27%	217	14%
LUFact	239	240	0%	328	37%	262	10%
Series	70	71	1%	115	64%	79	13%
SOR	56	72	29%	155	176%	110	96%
SparseMatmult	60	100	67%	109	82%	74	23%
MD	261	-	-	283	8%	271	4%
RayTracer	240	240	0%	273	14%	259	8%

Implementation with AspectJ

- Based on AspectJ code templates and a pre-processing tool

```
//CondExe<'void A.m()','getId()==0'>
void around(A obj) :
    call(void A.m())
    && target(obj) {
  if(getId()==0) proceed(obj);
}

//OneWay<'void A.m()'>
void around(): call(void A.m()) {
  Runnable t = new Runnable() {
    public void run() {
      proceed();
    }
  };
  submitTask(t);   // submit task for execution
}
```

- New templates can be added (requires AspectJ knowledge)

Discussion

- Assessment of OpenMP, MPI and pluggable parallelisation

	OpenMP	**MPI**	**PP**
localised / modular parallelisation	no (yes)	no	yes
incremental parallelisation	yes (no)	no	yes
unpluggability	yes	no	yes
code reuse / composition of abstractions	no	no	yes
support for new abstractions	no	no	yes
support for multi-core/ cluster/grids	yes/no/ no	no/yes/no	yes/yes/yes

Concluding remarks

- Pluggable parallelisation provides:
  - Separation of concerns
  - Incremental development
  - Support for explicit parallelisation of “legacy code”
- Performance is close to hand-written parallel code
  - Better composability and extensability
  - Less NCSS than with MPI
- Future Work
  - Start with non-code representations