Coherent Parallel Programming in C//

Zhiwei Xu
National Research Center for Intelligent Computing
Chinese Academy of Sciences
P.O. Box 2704
Beijing, China
zxu@apple.ncia.ac.cn

Kai Hwang
Department of Computer Science
University of Hong Kong
Pokfulam, Hong Kong
kaihwang@hkueee.hku.hk

Abstract: This paper presents the coherent parallel programming concept using a new parallel language called C// (pronounced C Parallel). The C// language is based on the standard C language with a small set of extended constructs for parallelism and process interaction. At the core of C// is a structured construct called coherent region, which facilitates the development of coherent programs, i.e., parallel programs that are structured, determinate, terminative, and compositional. We present the basic features of C// and show that coherent region is a versatile construct.

1. Introduction

The last decade has seen remarkable advances in the development of scalable parallel computer hardware[3]. However, their efficient utilization is severely hindered by the lack of adequate parallel software, the development of which has lagged far behind. We believe a key to overcoming the parallel software crisis is a parallel language that facilitates the development of coherent programs, where coherent means the following properties: terminative, determinate, compositional, and structured.

(1) Terminative: the program always finishes execution and enters a final state, starting from any initial state. When a program is not terminative, it can be either divergent or deadlock. A program is divergent, if for some initial state, it keeps executing without being able to finish. A program deadlocks, if it is stuck in the middle of execution, without being able to continue execution and is not in a final state.

Nontermination manifests itself differently in sequential and parallel programs. Sequential programs do not deadlock. Thus if a sequential program is not terminative, it must be divergent. An example is the infinite loop "while (x>0) x:=x+1". In parallel programs, there is a second type of divergence, called livelock, where several processes are trapped in infinite loops due to mutual interference. A simple example is the following code:

P1: while (y>0) y:=y+1;
P2: while (x>0) y:=x+1;

Of course, a parallel program can also be non-terminative due to deadlocks. It would be nice if the compiler or the run time system can detect all nonterminations. Unfortunately, this problem is equivalent to Turing Machine halting problem, which is undecidable. Thus all we can hope for is that the parallel language is designed to either prevent deadlocks and livelocks, or to allow the compiler and the run time system to detect them.

(2) Determinate: A program is determinate, if for any data state σ, there is only one final data state. That is, the program is a one-to-one function over the set of data states. If a program is not determinate, we say it is indeterminate. Determinacy guarantees that the results of a program are reproducible, which facilitates debugging, among other things.

(3) Compositional: A program is compositional if its semantics can be uniquely determined by the program's structure and the semantics of its components. For instance, the semantics of if (C) R else T is completely determined by its structure "if (...) ... else ..." and the semantics of its components: C, R and T. Compositionaly brings many benefits. For instance, it is easier to understand a compositional program than a non-compositional one, since to understand the more complex composite program, we only need to understand its structure and the less complex components. Compositionaly also helps program development. For example, after a top-down refinement step, we can use compositionality to check if the components realize the desired semantics.

(4) Structured: A program is structured if it is composed of structured constructs, where each structured construct has the following properties:

- It can be considered as a single-entry-single-exit construct;
- Different semantic entities are clearly identified in the syntax of the construct;
- Related operations are enclosed in one construct, instead of being scattered among different places in the program.

We have learned in sequential programming that structured constructs are preferred to unstructured ones [230].

0-8186-7876-3/96 $10.00 © 1997 IEEE

116
The same is true for parallel programming. For instance, it is
harder to give formal semantics for unstructured con-
structs than for structured ones [33]. Data flow analysis of
structured sequential programs can be conducted in a
purely syntax-directed manner, which implies more effi-
cient compile time checking and optimization. However, if
unstructured constructs are allowed, we need more com-
plex methods, such as interval analysis (if the underlying
flow graph is reducible) or iterative analysis (for general
flow graphs).

This paper presents CII, a flexible parallel language
based on C language with a small set of extensions which
facilitates coherent parallel programming. We first present
in Section 2 the language constructs. In Section 3 we dem-
onstrate the flexibility of CII by showing how it can specify
various types of parallelisms and process interactions. We
then presents a number of formal conditions under which
CII programs are guaranteed to be coherent.

The following conventions are followed: Boldface
words indicate reserved words.

2. Specifying Parallelism

CII uses two new constructs to specify parallelism: par-
allel block and parallel loop. We assume the usual C def-
initions of variables and expressions. A C statement is also
called a process. A sequential block is a sequence of state-
ments enclosed in a pair of curly brackets: \{ \textbf{S}_1 \textbf{S}_2 \ldots \textbf{S}_n \} .
MPMD (\textit{multiple-progam-multiple-data}) parallelism is
specified using the following parallel block construct, which
extends the sequential block construct:

\[
\textbf{par} \ [\text{region\_name\_declarations}] \ \{ \textbf{S}_1 \textbf{S}_2 \ldots \textbf{S}_n \}
\]

where each component process \textbf{S}_j can be any process, such
as another parallel block. The optional region name decla-
ration will be discussed shortly. This construct is similar to
Dijkstra’s cobegin, where the \textit{n} processes start execution
simultaneously when the parallel block is executed. A par-
allel block terminates when all its component processes
terminate.

We also have a short-hand notation, the parallel loop
construct for specifying SPMD (single-program-multiple-
data) parallelism, which extends the sequential for loop
construct:

\[
\textbf{parfor} \ (i=1;e2;e3) /* parallel loop head with loop
index variable \textit{i} */
\]

\[
[\text{region\_name\_declarations}]
\]

\[
\{ \textbf{S} \} /* loop body */
\]

The expressions \textit{e1}, \textit{e2}, and \textit{e3} in the loop head are used to
generate a set \( I = \{ I_1, I_2, \ldots, I_n \} \) of loop index values,
in a way similar to the sequential for loop:

\[
I = \varnothing ; \ j = e1; \ \textbf{while} \ (e2) \ \{ \ I = I \cup \{i\} ; e3; \}
\]

Let \( n(I) \) (or simply \textit{n}) denote the number of elements of
\( I \). The parallel loop construct creates \( n(I) \) processes, where
each process is a copy of the identical code \textit{S} and each
uses a different loop index value. Thus a parallel for loop
can be viewed as a short-hand notation for an equivalent
parallel block. In general, the parallel loop

\[
\textbf{parfor} \ (e1;\ e2;\ e3) \ \{ \ \textbf{S} \}
\]

is equivalent to

\[
\textbf{par} \ \{ \textbf{S}(I_1) \} \ \{ \textbf{S}(I_2) \} \ \ldots \ \{ \textbf{S}(I_n) \} \}
\]

For instance, parallel block \textbf{par} \ \{Process(1) \ldots Process(n)\}
is equivalent to \textbf{parfor} \ (i=1; \ i<=n; \ i++) \ \{ \ \text{Process} \ (i) \},
where \( I = \{1, 2, \ldots, n\} \).

Remarks The loop head is evaluated to generate the
complete index value set \( I \) before any loop body statement
is executed. \textit{i} should always be finite, otherwise the par-
allel loop will not terminate, as in the case of \textbf{parfor} \ (i=1; \ i++; \ \textit{i++)} \ \{ \ldots \}. If \textit{I} is empty, the parallel loop is equivalent to
\textit{un} empty statement `;'; \textit{I} should be evaluated from the
loop head alone. To avoid confusion, CII requires that \textit{e1}
and \textit{e2} be side-effect free, and that \textit{e3} can only affect the
loop index variable.

3. Specifying Interaction

The component processes of a parallel block (or a par-
allel for loop) interact through calls on coherent regions.
Such a region call starts with a reserved keyword \textbf{core}, fol-
lowed by an optional \textit{region name}, an optional \textit{region con-
dition}, and a \textit{region body} which is enclosed in two barriers:

\[
\textbf{core} \ \textbf{region\_name} \ \textbf{region\_condition}
\]

\[
\{ \textbf{region\_body} \}
\]

\[
/* entry barrier */
\]

\[
/* exit barrier */
\]

Region Name A region may or may not have a name.
Any region with a name must have the region name
declared at the beginning of a \textbf{par} block (or parallel loop)
as follows:

\[
\textbf{par}
\]

\[
\textbf{region} \ \textbf{region\_name} \ \textbf{[ participant\_list ]} ;
\]

\[
\{ \ldots \} \]

where the optional participant list enumerates all processes
(called the \textit{participants} of the region) that use the region to
interact. When this list is not explicitly specified, all com-
ponent processes of the parallel block are its participants.
A region \textit{R} should always be called by all its participants
through a region call statement \textbf{core} \textit{R}.

When a region name is declared in a parallel for loop,
its participant list can also be specified by a condition of
the loop index. A process \textbf{S}(\textit{i}) is a participant if \textit{i} satisfies
the condition. For instance, consider the following loop,

```parfor (i=0; i<6; i++)
region R1 (i % 2 == 0); R2 (i % 2 == 1);
{S(0)};
```

Here participants(R1) = {S(0), S(2), S(4)} and participants(R2) = {S(1), S(3), S(5)}.

The scope of a region name is the body of the `par` block or `parfor` loop where it is declared, less any function call in the body. In other words, processes inside a function (or procedure) can not interact with processes outside through named regions. A region may have no name. The unique participant of a `nameless` region is the process which executes the region call `core [(Region-Condition)]{...}`.

**Region Condition** A region condition is a logic expression without side effect. When the region condition is missing, it is considered as if the condition expression were always TRUE. When a named region is called by its participants, the conditions in all the region calls of its participants must be identical. Thus we are justified to say the condition of a region.

**Region Body** A region body consists of a sequence of zero or more segments. There are of three types of segments: `forall`, `across`, and `sequential`. A `parallel segment` refers to either a `forall` segment or `across` segment. An `across` or a sequential segment must be tagged with an `across` or a `seq` reserved word. If a `forall` segment is the first segment in the region body, its tag may be missing. Otherwise it must be tagged with a `forall` reserved word.

At either compile time or run time, the region bodies in all participants of a region R can be (conceptually, not necessarily in implementation) combined to form a single region body. Assume there are n participant processes P1, ..., Pn where the region body in Pi has the form:

```{ all forall segment-i
    across across segment-i
    seq sequential segment 
 }
```

Then region bodies can be combined into a single body as follows:

```{ all { forall segment-1 ... forall segment-n }
    across { across segment-1 ... across segment-n }
    seq { sequential statement }
 }
```

A `sequential segment` contains a sequence of (conditional) assignment statements. The sequential segments of the same region from all participants must have identical code. Otherwise, an error will be reported. Each `forall` or `across segment` contains a sequence of the following statements:

- Assignment.
- Aggregation assignment of the form "var = op (expression)!", where op is a reduction or scan operation, such as "sum", "minimum", etc.
- Conditional assignment of the form 'if (condition) assignment' or 'if (condition) combining assignment'. The condition should not have any side effect.

Each parallel or across segment must satisfy the `single-assignment rule`, which says that no variable can be assigned more than once in a segment.

**The Semantics of Coherent Region** The semantics of a region call can be understood as follows: When a process reaches a `core` R, it has to wait until all participants of the region R also reach their `core` R. Then the region condition for this region is evaluated. If the condition evaluates to FALSE, all participant processes wait until the condition becomes TRUE. Then all participants enter the region in parallel to execute the region body. After the body is executed, all participants leave the region to execute their respective subsequent computations.

The evaluation of the region condition and execution of the region body is viewed as an `atomic` action. The body of a region (we assume all bodies of a region are combined into a single region) is executed according to the following rules:

- If there are multiple segments in a region body, they should be executed one after another in the order specified.
- The sequential segment is executed as a sequential block.
- All assignments in a parallel segment can be viewed to conceptually form a parallel assignment statement with the following syntax:
  ```V1, ..., Vn = Expression-1, ..., Expression-N```
  The semantics is that all N expressions are first evaluated to N values. Then the N values are assigned to the N corresponding variables.
- An `across` segment is executed according to the data driven semantics: an operation can be performed when all its operands become available. The availability of operands is defined as follows: Constants are always available. Variables not appearing in the left-hand side of any assignment in the segment are always available. A variable appearing in the left-hand side of an assignment becomes available when it is assigned in the segment. Consider the following code for computing a recurrence equation:

```parfor (i=1; i<=2; i++)
region R;
{core R { 
     across
}
```
S1(i):c[i] = a[i]*b[i];
S2(i):x[i] = x[i-1] - c[i] * b[i];
}

The data-driven semantics of this program dictates that S2(1) executes after S1(1), and S2(2) executes after S1(2) and S2(1).

Note that to facilitate parallel execution and determine computation of a region, CII requires single assignment of forall and across segments. That is, no elementary variable can be assigned in the same segment more than once, where an elementary variable is any scalar variable or any array element variable. For an across segment, CII further requires that no elementary variable should depend on itself. This is allowed in a forall segment because of the parallel assignment semantics. CII's use of single assignment is different from that in a dataflow language such as SISAL or Id in the following ways:

- CII allows assignment to individual array elements;
- CII requires single assignment only for forall and across segments, where it is most needed, not the entire program, as in a dataflow language;
- Single assignment is required with respect to elementary variables. In a dataflow program, assignments to two different elements of the same array will cause single assignment violation.

CII's special single assignment rules are designed to retain an imperative semantics, thus avoiding difficulties in handling arrays as found in dataflow languages. This benefit comes with a price, though. Unlike a dataflow language, single assignment in CII can not be completely checked at compile time. This is not a great loss, however. Compile time checking is possible for those assignments where the left elementary variables are either scalar or have compile-time determinable index expressions. If the compiler is not sure, a warning will be issued. The runtime system can generate an error if indeed there is a single-assignment violation.

4. Supports for Interactions

There are three types of interactions in a parallel program: communication passes data values among processes; synchronization forces one or more processes to wait; and aggregation combines the partial values from individual processes to generate final results. Coherent region in CII is a very flexible construct. Combinations of the three orthogonal concepts (region name, region condition, and region body) can be used to specify various interactions.

CII assumes the shared-memory model: communication is through referencing shared variables. For instance, CII can use a named, unconditional region with a forall segment body to specify the communication mode used in all theoretical parallel algorithms that are based on the PRAM models. The following code fragment demonstrates how broadcast, multicast, permutation, and conditional communications can be realized in CII:

```c
parfor (i=0;i<n;i++)
  region Region1;
  {
    core Region1
    [ a[i] = d ; /* a 1-to-n broadcast */
      b[i] = c[f[i]] ; /* an m-to-n multicast */
      if f[i] evaluate to m values *
      c[i] = c[i+1 mod n]; /* a permutation */
      /* even-numbered processes send u[i] to neighbors */
      if (i % 2 == 0) left[i+1]=right[i-1]=u[i];
    } ... 
  }
```

We classify synchronization operations into three types: atomicity, data synchronization, and control synchronization. All can be supported by CII. An atomic operation is specified by an unnamed, unconditional region, e.g.,

```c
parfor (i=1;i<n;i++) {
  core { x= x+1; y= y-1; } /* the two assignments
      are executed as a single atomic operation */
}
```

A process executing a control synchronization operation will wait until the program execution reaches certain control states. A common example of control synchronization is the so called barrier synchronization, as shown in the following code:

```c
parfor (i=1;i<n;i++)
  region R;
  {
    P_i
    core R () /* a barrier */
    Q_i
  }
```

Here we have n processes. The i-th process executing $P_i$ followed by a barrier core R () followed by $Q_i$. When it finishes $P_i$ and reaches the barrier, it has to wait until all other processes also reach their barrier. Another control synchronization construct is the critical region, as illustrated below:

```c
parfor (i=1;i<n;i++) {
  core { seq x= x+1; y= y-1; }
}
```

Note that a critical region is mutual exclusive, in that only one process is allowed to execute the two assignments in the region body at a time. In other words, the region body
is executed sequentially. In contrast, multiple processes can execute their atomic regions, as long as the atomicity is enforced.

A process executing a data synchronization operation will wait until the program execution reaches certain data states. For instance, a process executing a \texttt{wait(x>0)} statement will be delayed until variable x becomes positive. Examples of data synchronization operations include locks, conditional critical regions, monitors, and events. In most current systems, atomicity is realized through data synchronization, such as the following:

\begin{verbatim}
parfor (i=1; i<n; i++) {
    lock(S);
    x = x+1; y = y-1;
    unlock(S);
}
\end{verbatim}

where the lock synchronization depends on the data state in semaphore S. Various data synchronization operations can be realized by \texttt{C/C} conditional coherent regions, as shown in Fig. 1.

Control synchronization only depends on the program's control state, and is not affected by the program's data state. Control synchronization is generally easier to understand than data synchronization, although the later is more flexible. In \texttt{C/C}, these two types of synchronizations are specified differently.

We say a competitive interaction is involved when processes interact by competing to a shared resource. All others interactions are called cooperative. Usually, cooperative interactions are realized by named unconditional regions, while competitive interactions are realized by nameless conditional regions. However, some interactions can be best described as both competitive and cooperative, and are best handled by named conditional region. As a general rule, competitive interactions are realized in \texttt{C/C} through the use of a conditional coherent region. In most cases (as demonstrated in Fig. 1.), a nameless conditional region with a very short region body is enough.

An aggregation operation is used to merge partial results computed by the component processes of a parallel program to generate a complete result. Its main characteristic is that it can be realized as a sequence of supersteps, and each superstep consists of a short computation and a simple communication and/or synchronization. As an example, consider the following program which computes the inner product of vectors A and B:

\begin{verbatim}
parfor (i=0; i<n; i++) {
    x[i] := A[i] * B[i];
    inner_product = aggregate_sum(x[i]);
}
\end{verbatim}

Here \texttt{aggregate_sum} is an aggregation operation, which merges the partial results $x[0]$, $x[1]$, ..., $x[n-1]$ to generate a final result $\text{inner\_product}=x[0]+x[1]+...+x[n-1]$. This summation operation is called a \textit{reduction}, which reduces a number of values into a single value. Other types of aggregations include \texttt{scan} (parallel prefix), \texttt{descend algorithms}, \texttt{ascending algorithms}, etc.

\texttt{C/C} allows user-definable aggregation operations. An aggregation is first defined using an aggregation block, and then used in coherent regions. An aggregation needs to be defined only once in a program, but can be used in multiple coherent regions many times. The following shows a reduction block that defines reduction operations:

\begin{verbatim}
reduction (x,y) /* This reduction block defines one or more reduction operations, where the variables x and y are in a domain D */
{
    F ::= singleton is expr1 /* F([x]) returns expr1 */
    function is /* F([x,y]) is computed by */
    B(F(x),F(y))) /* with parameters F(x) and F(y) */
    sum::= /* a missing singleton definition implies sum(x) is x, and the function is phrase is also omitted. */
    \{ x+y \} /* sum([x,y]) returns x+y */
    max::= /* a missing singleton definition implies max(x) is x */
    \{ (x > y) ? x : y \} /* max([x,y]) returns the larger of x and y */
    count::=singleton is (int 1) /* count([x]) returns integer 1 */
    function is /* count([x,y]) returns count(x)+count(y) */
    \{ count(x) + count(y) \}
}
\end{verbatim}

In the definition of a reduction \( F, \text{expr1} \) can be any expression with \( x \) as the only variable; and the function body can be either an expression (e.g., in \texttt{sum} and \texttt{count}) or a procedure which returns a value (e.g., in \texttt{max}). Again the only parameters allowed in a function body are \( F([x]) \) and \( F([y]) \), which can be shortened to \( F(x) \) and \( F(y) \). The definition of a reduction operator can be simplified, once we know that \( F([x]) = x \) (which is satisfied by many practical reduction operations). In the above reduction block, the function bodies of sum and max are simplified (\( F(x) \) and \( F(y) \) are replaced with \( x \) and \( y \), respectively). Since in both operations, we have \( F([x]) = x \). (In fact, we use this as a default rule so that we do not have to explicitly specify the
singleton definition.) However, this simplification does not extend to the reduction operator "count", where count([x]) = x does not hold.

For a reduction operation \( F, F([x]) \) and \( F_R \) are defined in the reduction block, shown above in the singleton and the function definitions. Domain \( D \), operand list \( X \), and the list length \( n \) are determined by the system when \( F \) is used in a parfor or par block:

```c
int i, a[N], b;
float c[N], d;
parfor (i=0;i<N;i++)
region R;
{compute a[i] and c[i];
core R {
b = sum(a[i]);
d = sum(c[i]);
}
}
```

When a reduction \( F \) is used, it is used in a reduction assignment with the form \( \text{var} = F(\text{expr}) \). The parfor head "parfor (i=0;i<N;i++)" and \( \text{expr} \) determines the domain \( D \), the operand list \( X \), and the list length \( n \). In the first reduction assignment in the above parfor loop, \( F \) is sum, \( \text{expr} \) is \( a[i] \), \( D \) and \( R \) are the integer type, \( n = N \), and \( X = [a[0], a[1], ..., a[N-1]] \). Although the second reduction assignment uses the same reduction operation "sum", \( D \) and \( R \) are now the float type, and \( X = [c[0], c[1], ..., c[N-1]] \), because \( \text{expr} \) is now \( c[i] \). More specifically, \( X, n, D, \) and \( R \) are determined as follows:

- The system (either the compiler or the run time system) determines a sequence of indexes \( i_1, i_2, ..., i_N \) from \( e_1, e_2, \) and \( e_3 \) of the parfor head "parfor (i=0;i<N;i++)", which can be used, together with \( \text{expr} \) in a reduction assignment "\( \text{var} = F(\text{expr}) \)" to determine \( n = N \) and \( X = [\text{expr}(i_1), ..., \text{expr}(i_N)] \).
  - \( D \) is the data type of \( \text{expr} \).
  - \( R \) is determined by the singleton definition "singleton is \( expr \)". That is, \( R \) is the data type of \( expr \), which may depend on \( D \).

A cooperative interaction with more complex patterns can usually be realized with several segments in the body of a named region. This is exemplified by the following parallel block, which computes the histogram, \( \text{hist}[i] \) (for \( i = 0, 1, ..., \text{binnumber}-1 \), of an \( n \)-dimensional array \( A \). It demonstrates the use of forall, across, and sequential segments, as well as the use of reduction operators.

```c
parfor (i=0;i<N;i++)
region R;
{core R /* Assume reduction operators max, min, and count are already defined in a reduction block before */

amax = max (A[i]); /* Use reductions to find the maximum element */
amin = min (A[i]); /* and the minimum elements of array A */

seq
binspace = (amax - amin) / binnumber;

across
bin[i] = 1 + (a[i] - amin) / binspace;
\text{hist}[bin[i]] = \text{count}(bin[i]); /* use reduction operator count */
}
```

5. Coherent Program Classes in C++

The Write-Restriction Rule is a structuring rule stating the following two conditions: (1) All writes to shared variables must be in a region body; and (2) No shared variable can be written in two regions that can execute in parallel. The Reducibility Rule is a structuring rule which states the following two conditions: (1) If there is a named region call within a while loop, then all region calls with the same region name must also be contained within the same while loop; and (2) All named regions are partially ordered. We have defined a formal semantic model of C++ based on Petri nets. The following is a general result:

**Theorem 1.** Any program satisfying the Reducibility Rule has a reducible Petri net, and vice versa (where the reducibility of a Petri net is similarly defined as in a flow graph[1], i.e., after deleting all back edges, the graph is acyclic).

Some results are also obtained with respect to the following classes: The UNR class allows only Unconditional, Named Regions. There is no restriction on the region body, nor is there any structuring rule. The Write-Restricted UNR (or WRUNR) class is obtained by adding to the RUN class the write-restriction rule.

**Theorem 2.** If a program of the UNR class is not terminating, either it has infinite looping, an irreducible Petri net, or confusion[4] is possible in the net.

**Theorem 3.** A terminating UNR program is determinate, if for each of its parallel blocks, any two statements in any maximum antichain satisfy Bernstein's condition[7].

**Theorem 4.** Any terminating WRUNR program is determinate and compositional.

Note that even the seemingly restrictive WRUNR class can code all parallel algorithms based on the PRAM models. The only known class of imperative parallel programs
that are guaranteed to be determinate and compositional is
the class of disjoint parallel programs[2], i.e., programs in
which parallel processes do not interact. Thus, Theorem 5
is a meaningful generalization of known results, as pro-
cesses are allowed to interact via named regions.

6. References


```
LOCK(S)
critical section
UNLOCK(S)
(a) mutual exclusion using locks

await(S);
signal(S);
clear(S);
(c) the three event operations

await (condition) do critical_section
(e) conditional critical region

result = F&A(x, v)
(g) a fetch-and-add operation

C&S(w, old, new)
int * w, old, new;
{ if (*w==old) { *w=new; return 1; }
  else return 0;
}
(i) compare and swap

core R {}
(k) barrier synchronization in C/

core (S) { S=S-1; }
critical section

core { S=S+1; }
(b) C/ code for mutual exclusion

core (S){}

core {S=1;}

core {S=0;}
(d) C/ codes for the three event operations

core (condition) { seq critical_section }
(f) C/ code for conditional critical region

core { result=x; x=x+v; }
(h) C/ code for fetch-and-add

core {
seq
result=(w==old)?w=new,1:0;
}
(j) C/ code for "result=C&S(&w,old,new)"

core R { s = Red_Sum(a[i]) ; }
(l) sum reduction in C/
```

Figure 1. Realization of various interactions by coherent regions in C/