APPENDIX

Cilk Plus

B

This appendix provides a concise introduction to the features of Intel Cilk Plus (Cilk Plus) with an emphasis on portions used within this book. For more complete information, see the Intel Cilk Plus Language Specification and Intel Cilk Plus Application Binary Interface Specification documents, which are available from http://cilkplus.org.

Figure B.1 outlines the components of Cilk Plus, as well as the parts of TBB that are recommended for use with it.

B.1 SHARED DESIGN PRINCIPLES WITH TBB

Cilk Plus and TBB purposefully share key attributes as parallel programming models: separation of programmer/tool responsibilities, composability including interoperability with other parallel programming models, support for current programming environments, readability by adhering to serial semantics, and portability of source code.

The key design principle behind effective parallel programming abstractions is that the programmer should identify elements that can be safely executed in parallel. Separately, the runtime environment, particularly the scheduler, should decide during execution how to actually divide the work between processors. This separation of responsibilities allows code to run efficiently on any number of processors, including one processor, without customizing the source code to the number of processors.

B.2 UNIQUE CHARACTERISTICS

Cilk Plus is distinguished by its focus on minimal but sufficient support for parallelism in C and C++. It is easy to learn, able to support sophisticated debugging tools, and provides guarantees that bound memory usage. Cilk Plus does this all while also scaling to high degrees of thread *and* vector parallelism.

Cilk Plus seeks to address shortcomings of a template library approach:

- Usability in C, not just C++
- Support for vector parallelism
- Serial elision, where a Cilk Plus program run with one thread behaves as if the Cilk Plus keywords are replaced with standard C/C++, as will be explained in Section B.4
- Conveying parallel structure to the compiler at a higher level than a template library, which enables more optimization opportunities

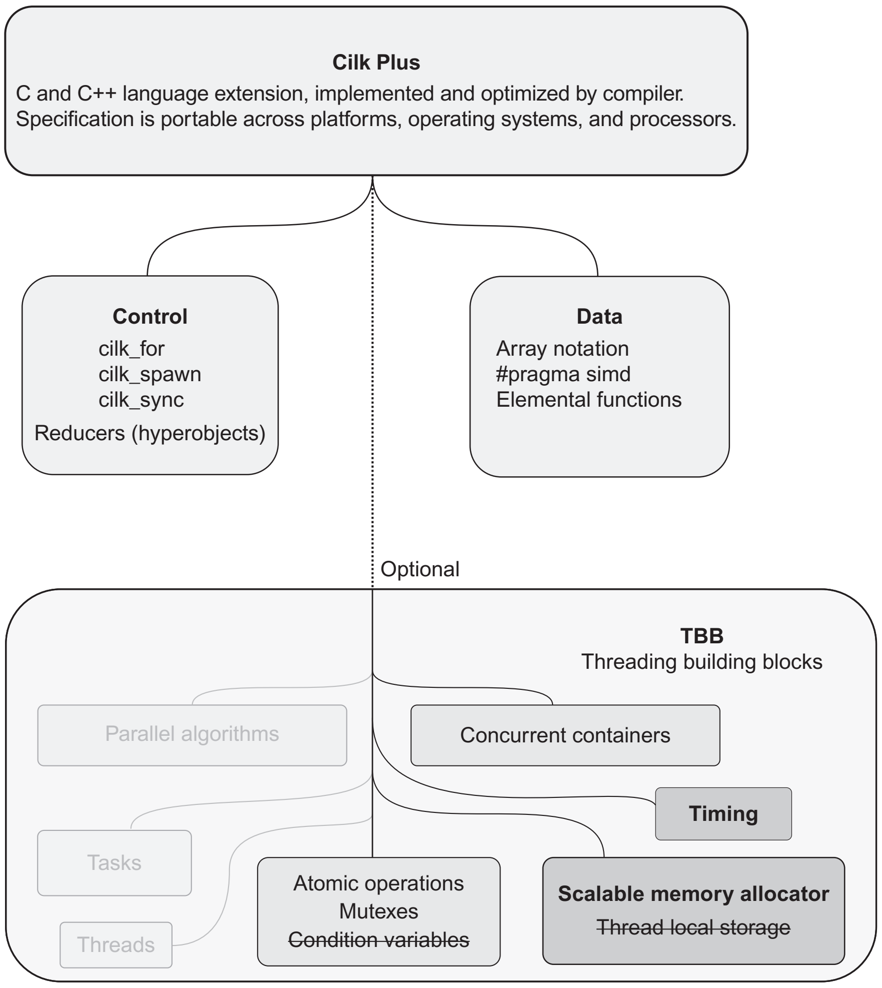

FIGURE B.1

Overview of Cilk Plus. Parts of TBB can be borrowed for foundation functionality. Darkly shaded TBB components are recommended for use with Cilk Plus. Lightly shaded TBB components interoperate with Cilk Plus but may break determinism. Faded components also interoperate but are essentially alternatives to Cilk Plus. Crossed-out portions carry risks. Section B.3 details our recommendations.

These items require compiler support and are therefore beyond the scope of a template library such as TBB.

Because Cilk Plus requires implementation inside a compiler, it is not yet as widely portable as TBB. Thus, to promote adoption of Cilk Plus and make it as widespread as TBB, Intel is making it easy for compiler makers to implement it. Cilk Plus is published as an open language specification *and* an open ABI specification. Furthermore, Intel is working on making all key components open source.

Currently, a portable form of the runtime library is open source, along with a development branch of the GCC compiler that supports `cilk_spawn`, `cilk_sync`, `cilk_for`, reducers, `#pragma simd`, and array notation. The open source version is expected to eventually become a complete Cilk Plus implementation.

B.3 BORROWING COMPONENTS FROM TBB

Cilk Plus does not duplicate functionality that can be borrowed from TBB. Indeed, we encourage Cilk Plus programmers to use components of TBB that are orthogonal to expression of parallelism. These components of TBB are:

- Scalable memory allocator
- `tick_count` timing facility

Some portions of TBB are all right to use but are not the Cilk Plus ideal because they break determinism or greedy scheduling theory:

- Mutexes
- Atomic objects
- Concurrent containers

Consider using alternative solutions based on Cilk Plus **hyperobjects** if you can to reap the benefits of determinism.

The parallel algorithms and tasks in TBB can interoperate with Cilk Plus, but using them instead of Cilk Plus forgoes the key value proposition of Cilk Plus.

We discourage combining two of TBB's features with Cilk Plus's control-flow constructs. The feature combinations to avoid are:

- Using condition variables with `cilk_spawn` or `cilk_for`. This is questionable since the purpose of a condition variable is to wait for *another* thread to change state protected by a mutex. Since Cilk Plus task constructs only *permit* thread parallelism, but do not *mandate* it, there might not be another thread. The wait could last forever.
- Mixing thread-local storage with Cilk Plus's `cilk_spawn` or `cilk_for`. This invites trouble. The problem is that Cilk Plus is not about threads. It is about running parallel **strands** of execution, where a strand (Section 8.10) is a segment of control flow with no intervening fork or join point. However, the runtime maps strands to threads in ways that can surprise neophytes. For example, when a worker thread created by the Cilk Plus runtime calls a function, the function can return on a *different* thread.

Hyperobjects are usually an excellent alternative to thread-local storage. The k-means example (Chapter 11) lets you compare these alternatives. The TBB code uses thread-local storage; the Cilk Plus code uses two hyperobjects.

Borrowing can go the other way, too. The Cilk Plus features for vector parallelism (array notation and `#pragma simd`) are an excellent way to exploit vector parallelism in a TBB program, even if the tasking model supported by Cilk Plus is not used. Note that if you express vector parallelism using only **elemental functions** and the #pragma simd, the code will still be portable to other compilers, since ignoring these constructs still gives the expected result.

B.4 KEYWORD SPELLING

This book's spelling of the keywords requires inclusion of the header <cilk/cilk.h>, which has:

```
#define cilk_spawn _Cilk_spawn
#define cilk_sync _Cilk_sync
#define cilk_for _Cilk_for
```

The compiler recognizes only the keywords on the right. The reason is that the introduction of new keywords by a compiler is limited, by convention and standards, to names beginning with an underscore followed by an uppercase letter. Such symbols are reserved to compiler implementers and should never cause a conflict with application code. The header provides more aesthetically pleasing spellings.

Including the <cilk/cilk_stub.h> header file converts a program to its **serial elision** by defining:

```
#define _Cilk_spawn
#define _Cilk_sync
#define _Cilk_for for
```

These substitutions revert the program to a serial program that can be compiled by any C++ compiler. The resulting code will behave just like a Cilk Plus program running on a single thread.

B.5 cilk_for

The syntax for cilk_for is similar to for. It looks like:

cilk_for (*initialization*; *condition*; *increment*) *body*

The *body* can be a single statement or a block of code. Changing a for loop to a cilk_for loop permits the iterations to run in parallel.

A cilk_for loop has the following constraints that do not exist for a for loop:

- Control may not be explicitly transferred out of the body or into it. In particular, return and break are prohibited. A goto must not jump from inside the body to outside of it, or vice versa.
- Control may be implicitly transferred out of the body by an exception. In that case, which other iterations execute depends on the implementation and might not be deterministic. The exception

```
cilk_for (int i=ivalue; i<limit; ++i) {
   a[i] = foo(b[i],c[i]) * 3.0;
}
```

LISTING B.1

Simple example use of cilk_for.

thrown from the `cilk_for` is the same as the serial elision would have thrown, even if multiple iterations throw.

- The *initialization* shall declare or initialize a single variable only. This is called the *control variable*. In C, the control variable may be declared outside the `cilk_for` loop. In C++, the initialization must declare the control variable. The variable cannot be declared `const` or `volatile`.
- The control variable may not be modified within the loop body.
- The *increment* must have one of the following forms:

  i++
  ++*i*
  i−−
  −−*i*
  i+=*step*
  i−=*step*

  where `i` stands for the control variable, and `step` can be any expression.
- The *condition* must consist of the control variable *i* compared with another expression, which we will call the *limit*. The comparison can be =>, >, <=, <, !=, or ==.
- The *step* (if any) and *limit* expressions must not be affected by the loop body. This is so that the number of iterations can be computed correctly before any iterations commence. For C programs, the value of the control variable, if declared outside the loop, has the same value after a `cilk_for` loop as it would have after a `for` loop.

The language extension specification [Cor11b] explains the details more precisely, with attention to fine points and should be consulted for a complete definition.

B.6 cilk_spawn AND cilk_sync

The `cilk_spawn` keyword specifies that the caller of a function may continue to run without waiting for the called function to return.

```
// Simple function call
cilk_spawn bar(1);
// Lambdas allowed
cilk_spawn []{ bar(2); }();
// Results allowed
result = cilk_spawn bar(4);
// Innermost call completes before spawn
result = cilk_spawn bar( bar(9) );
// Spawn not used, no need, potentially wasteful
bar(5);
// Wait for all spawns
cilk_sync;
```

LISTING B.2

Examples of using `cilk_spawn` and `cilk_sync`.

Execution of a statement with the `cilk_spawn` keyword is called a **spawn**. The function, try block, or `cilk_for` body that contains the spawn is called the **spawning block**. Note that compound statements containing a spawn are *not* spawning blocks unless they fit one of the categories above.

Execution of a `cilk_sync` statement is called a **sync**. A sync waits only for spawns that have occurred in the same spawning block and has no effect on spawns done by other tasks or, done by other threads, nor those done prior to entering the current spawning block. An implicit sync occurs when exiting the enclosing spawning block. Thus, when a spawning block completes, any parallelism that it created is finished. This property simplifies reasoning about program composition and correspondence with its serial elision.

The following snippet illustrates some of these points:

```
void foo() {
   for (int i=0; i<3; ++i) {
      cilk_spawn bar(i);
      if (i%2) cilk_sync;
   }
   // Implicit cilk_sync
```

The snippet has one spawning block: the function body. The body of the `for` loop is *not* a spawning block because it is not the body of a function, try block, or `cilk_for`. The code operates as follows:

1. Spawn `bar(0)` and `bar(1)`.
2. Execute a sync that waits for the spawned calls.
3. Spawn `bar(2)`.
4. Execute the implicit `cilk_sync`.

The scope of the explicit sync is dynamic, not lexical. It applies to all prior spawns by the current invocation of `foo()`, since that is the innermost spawning block.

Jumping into, or out of, a spawning block results in undefined behavior. This includes use of `goto`, `setjmp`, and `longjmp`. "Undefined behavior" is a term of art in language specifications that means *anything* could happen, including crashing your program or turning your computer into a frog. You have been warned.

Behavior is defined if a spawned function throws an exception and does not catch it. The exception is rethrown when execution leaves the corresponding sync. If there are multiple such exceptions for the sync, the sync rethrows the exception that the serial elision would have thrown. The extra exceptions are destroyed without being caught.

B.7 REDUCERS (HYPEROBJECTS)

A hyperobject enables multiple strands of execution to operate in parallel on the same logical object, without locking, yet get the same result as the serial elision would get. Furthermore, a hyperobject avoids contention bottlenecks, because parallel strands get separate local views of the logical object. Section 8.10 discusses the theory of hyperobjects in more detail.

A **reducer** is a hyperobject intended for doing reductions. Cilk Plus reducers work for any operation that can be reassociated. Cilk Plus predefines reducers for common associative operations:

Operation	Header
list accumulation	`<cilk/reducer_list.h>`
min	`<cilk/reducer_min.h>`
max	`<cilk/reducer_max.h>`
addition and subtraction	`<cilk/reducer_opadd.h>`
bitwise AND	`<cilk/reducer_opand.h>`
bitwise OR	`<cilk/reducer_opor.h>`
bitwise EXCLUSIVE OR	`<cilk/reducer_opxor.h>`
string concatenation	`<cilk/reducer_string.h>`
reducer version of `std::ostream`	`<cilk/reducer_ostream.h>`

The last item might seem surprising, since output operations are not normally considered associative. However, a reducer can change operations to reassociate them. For example:

```
(cout << x) << y
```

can be reassociated as:

```
cout << (TOSTRING(x) + TOSTRING(y))
```

where `TOSTRING` denotes conversion to a string and + denotes string concatenation. Section 9.4.2 explains this in more detail. One other reducer mentioned also changes operations: The reducer `reducer_opadd` reassociates subtraction by rewriting $a - b$ as $a + (-b)$.

Another kind of hyperobject is a **holder**. Cilk Plus predefines holders in header `<cilk/holder.h>`. A holder is a kind of reducer where the reduction operation $\otimes$ does one of the following:

Policy	Operation
Keep last	$x \otimes y = y$
Keep indeterminate	$x \otimes y =$ arbitrary choice of x or y

A *keep-last holder* is useful for computing the last value the hyperobject would have after serial execution. It can be thought of as using the C/C++ "comma operator" for reduction. A *keep-indeterminate holder* is useful for holding race-free temporary storage instead of reallocating/freeing it every time a strand needs it. See comments in `<cilk/holder.h>` for more details, options, and examples for holders.

B.7.1 C++ Syntax

Our example uses `reducer_opadd` and is illustrated in Listing B.3. To use a predefined reducer:

1. Include the appropriate reducer header file. Our example uses `#include <cilk/reducer_opadd.h>`.

```
#include "cilk/reducer_opadd.h"
using namespace std;

void cpp_serial( int data[], size_t n ) {
   int result = 47;

   cout << "C++ reduction with for" << endl;

   for (size_t i = 0; i < n; ++i) {
      result += data[i];
   }

   cout << "Result is: " << result << endl;
}

void cpp_parallel( int data[], size_t n ) {
   cilk::reducer_opadd<int> result(47);

   cout << "C++ reduction with cilk_for" << endl;

   cilk_for (size_t i = 0; i < n; ++i) {
      result += data[i];
   }

   cout << "Result is: " << result.get_value() << endl;
}
```

LISTING B.3

Serial reduction in C++ and equivalent Cilk Plus code. See Listing B.4 for an equivalent example in C.

2. Declare the reduction variable as a `reducer_kind<TYPE>` rather than as a `TYPE`. The default value is the identity element of the reduction operation. If you need a different initial value, use a parenthetical expression, not = for the initializer. The = syntax will not work.[1] Our example uses:

   ```
   cilk::reducer_opadd<int> result(47);
   ```

   not this:

   ```
   cilk::reducer_opadd<int> result=47; // WRONG!
   ```

3. Introduce parallelism, such as changing a `for` loop to a `cilk_for` loop. Update the reduction variable (in our example, `result`) just like before. It is not necessary to worry that the hyperobject now provides a strand-local view of the variable. However, updates are restricted to the reduction

[1] Why? Because the C++ standard requires that to use the = syntax the class must have a public copy constructor, even if the compiler optimizes it away. Hyperobjects generally have private copy constructors.

operations supported by the hyperobject. For example, `reducer_opadd` allows only +=, −=, ++, and −−.

4. Retrieve the reducer's terminal value with method `get_value()` *after* all strands that update it sync. In our example, this is `result.get_value()`, and the strands synced when the `cilk_for` loop finished. Retrieving the reducer's terminal value before all strands sync may return only a partial result.

To illustrate these steps for C++, Listing B.3 shows routines for summing an array of integers. One routine uses `for` and the other uses `cilk_for` with a reducer. Otherwise, they are identical C++ code to illustrate equivalence.

The header `<cilk/reducer.h>` defines a generic reducer that you can use to define your own custom reducer. Section 11.2.1 walks through the mechanics of defining a custom reducer.

B.7.2 C Syntax

Our example uses `reducer_opadd` and is illustrated in Listing B.4. The steps are:

1. `#include` the appropriate header for the reducer. Section B.7 lists the types and header files for predefined reducers. Our example uses `#include <cilk/reducer_opadd.h>`.
2. Declare the reducer object using:

   `CILK_C_REDUCER_`*type*`(`*variable_name*, *variable_type*, *initial_value*`);`

   For example, to declare an addition reducer variable `result` of type `int` initialized to zero, use:

   ```
   CILK_C_REDUCER_OPADD(result, int, 0);
   ```

3. After the declaration of the reducer but before the first use, insert:

   `CILK_C_REGISTER_REDUCER(`*reducer_name*`);`

   to register the variable with the Cilk Plus runtime. This provides for proper initialization and memory clean-up. It is not strictly needed if the variable *reducer_name* is a global variable.
4. To access the value in a serial region, use the member `value` of the reducer. In our example, this is `result.value`. Operations on the member should be restricted to the appropriate operations only, but, unlike in C++, this restriction cannot be enforced by the compiler. Failure to obey this restriction can easily produce invalid results. The Cilk Plus documentation on reduction operations lists the allowed operations. For instance, for `OPADD` the only allowed operations are +=, −=, ++, and −−.
5. When accessing the value of the reducer from parallel strands, use `REDUCER_VIEW(`*reducer_name*`)` to access the value. In our example, this is `REDUCER_VIEW(result)`.
6. When the reducer is no longer needed, insert

   `CILK_C_UNREGISTER_REDUCER(`*reducer_name*`);`

   Just like the registration of the variable, this is not strictly needed if the variable is global.

To illustrate these steps for C, Listing B.4 adds an array of numbers together using `for` and `cilk_for` with reducers in otherwise identical C code to illustrate equivalence in this C code.

```
#include "cilk/reducer_opadd.h"

void c_serial( int data[], size_t n ) {
   size_t i;
   int result = 47;

   printf("C reduction with for\n");

   for (i = 0; i < n; ++i) {
      result += data[i];
   }
   printf("Result is: %d\n", result);
}

void c_parallel( int data[], size_t n ) {
   size_t i;
   CILK_C_REDUCER_OPADD(result, int, 47);
   CILK_C_REGISTER_REDUCER(result);

   printf("C reduction with cilk_for\n");

   cilk_for (i = 0; i < n; ++i) {
      result.value += data[i];
   }
   printf("Result is: %d\n", REDUCER_VIEW(result));
   CILK_C_UNREGISTER_REDUCER(result);
}
```

LISTING B.4

Serial reduction in C and equivalent Cilk Plus code. See Listing B.3 for an equivalent example in C++.

The header `<cilk/reducer.h>` has macros to assist defining your own custom reducer. See that header and the Cilk Plus documentation for details on how to use these macros.

B.8 ARRAY NOTATION

Cilk Plus extends C and C++ with array notation, which lets the programmer specify array sections and operations on array sections. Programming with array notation achieves predictable performance based on mapping parallel constructs to the underlying hardware vector parallelism, and possibly thread parallelism in the future. The notation is explicit and easy to understand and enables compilers to exploit vector and thread parallelism with less reliance on alias and dependence analysis. For example,

```
a[0:n] = b[10:n] * c[20:n];
```

is an unordered equivalent of:

```
for ( int i=0; i<n; ++i )
   a[i] = b[10+i] + c[20+i]
```

Use array notation where your operations on arrays do not require a specific order of operations among elements of the arrays.

B.8.1 Specifying Array Sections

An array section operator is written as one of the following:

[*first* : *length* : *stride*]
[*first* : *length*]
[:]

where:

first is the index of the first element in the section.
length is the number of elements in the section.
stride is the difference between successive indices. The *stride* is optional, and if omitted is implicitly 1. The *stride* can be positive, zero, or negative.

All three of these values must be integers. The *j*th element of an array section $a[i:n:k]$ is $a[i+j\cdot k]$ for $j \in [0,n)$.[2]

The notation *expr*[:] is a shorthand for a whole array dimension if *expr* has array type before decay (conversion to pointer type) and the size of the array is known. If either `first` or `length` is specified, then both must be specified. Examples include:

```
float x[10];
x[0:5];      // First five elements of x
x[5:10];     // Last five elements of x
x[1:5:2];    // Elements of x with odd subscripts
x[:];        // All ten elements of x
```

A scalar or array section has a *rank*. A scalar has rank 0. The rank of an array section $a[i:n:k]$ is one more than the rank of a. The rank of $a[i]$ is the *sum* of the ranks of a and i. The rank of i must not exceed one. Successive array section operators behave analogously to multiple C/C++ subscripts. The shape of a multidimensional section is a tuple of the section lengths. Examples:

```
int s;
int u[5], v[4];
int a[7][4];
int b[8][7][4];
x;                // rank=0, shape=<>
v[0:4];           // rank=1, shape=<4>
```

[2] Note to Fortran 90 programmers: The middle of the triplet is a *length*, not the last index.

```
a[0:7][0:4];       // rank=2, shape=<7,4>
a[0][0:4];         // rank=1, shape=<4>
a[0:7][0];         // rank=1, shape=<7>
b[0:5][0][0:3];    // rank=2, shape=<5,3>
u[v[0:4]];         // rank=1, shape=<4>
```

The last line subscripts array `u` (rank zero) with an array section (rank one), and the rank of the result is the sum of those ranks.

B.8.2 Operations on Array Sections

Most C and C++ scalar operations act elementwise on array sections and return an elementwise result. For example, the expression `a[10:n]−b[20:n]` returns an array section of length `n` where the *j*th element is `a[10+j]−b[20+j]`. Each operand must have the same shape, unless it is a scalar operand. Scalar operands are reshaped by replication to match the shape of the non-scalar operands. Function calls are also applied elementwise. Examples include:

```
extern float x[8], y[8], z[8];
extern float a[8][8];
x[0:8] = x[0:8] + y[0:8];     // Vector addition
x[0:8] += y[0:8];             // Another vector addition
x[0:8] = (x[0:8]+y[0:8])/2;   // Vector average
a[3][0:8] = x[0:8];           // Copy x to row 3 of a
a[0:8][3] = x[0:8];           // Copy x to column 3 of a
z[0:8] = pow(x[0:8],3.f);     // Elementwise cubes of x
std::swap(x[0:8],y[0:8]);      // Elementwise swap
x[0:8] = x[0:5];              // Error – mismatched shape
a[0:5][0:5] = x[0:5];         // Error – mismatched shape
```

The few operators that are not applied elementwise or have peculiar rank rules are:

- **Comma operator:** The rank of x, y is the rank of y.
- **Array section operator:** As described earlier, the rank of $a[i:n:k]$ is one more than the rank of a.
- **Subscript operator:** As described earlier, the rank of the result of $a[i]$ is the sum of the ranks of a and i. The j element of $a[k[0:n]]$ is $a[k[j]]$. Trickier is the second subscript in $b[0:m][k[0:n]]$. Both $b[0:m]$ and $k[0:n]$ have rank one, so the result is a rank-two section where the element at subscript i,j is $b[i][k[j]]$.

Note that pointer arithmetic follows the elementwise rule just like other arithmetic. A consequence is that $a[i]$ is not always the same as $*(a+i)$ when array sections are involved. For example, if both a and i have rank one, then $a[i]$ has rank two, but $*(a+i)$ has rank one because it is elementwise unary $*$ applied to the result of elementwise $+$.

Historical note: In the design of array notation, an alternative was explored that preserved the identity $*(a+i) \equiv a[i]$, but it broke the identity $(a+i)+j \equiv a+(i+j)$ when a is a pointer type and made the rank of $a+i$ dependent on the type (not just the rank) of a. It turns out that array notation must break one of the two identities, and breaking associativity was deemed the worse of two evils.

B.8.3 Reductions on Array Sections

There are built-in operations for efficient reductions of array sections. For example, `__sec_reduce_add(a[0:n])` sums the values of array section `a[0:n]`. Here is a summary of the built-in operations. The last column shows the result of reducing a zero-length section.

Operation	Result	If Empty
`__sec_reduce_add`	$\Sigma_i a_i$	0
`__sec_reduce_mul`	$\Pi_i a_i$	1
`__sec_reduce_max`	$\max_i a_i$	"$-\infty$"
`__sec_reduce_min`	$\min_i a_i$	"∞"
`__sec_reduce_max_ind`	j such that $\forall i : a_j \geq a_i$	unspecified
`__sec_reduce_min_ind`	j such that $\forall i : a_j \leq a_i$	unspecified
`__sec_reduce_all_zero`	$\forall i : a_i = 0 \; ? \; 1 : 0$	1
`__sec_reduce_all_nonzero`	$\forall i : a_i \neq 0 \; ? \; 1 : 0$	1
`__sec_reduce_any_zero`	$\exists i : a_i = 0 \; ? \; 1 : 0$	0
`__sec_reduce_any_nonzero`	$\exists i : a_i \neq 0 \; ? \; 1 : 0$	0

The "$-\infty$" and "∞" are shorthands for the minimum and maximum representable values of the type.

The result of a reduction is always a scalar. For most of these reduction, the rank of a can be one or greater. The exception is that the rank of a must be one for `__sec_reduce_max_ind` and `__sec_reduce_min_ind`. These return an index of type `ptrdiff_t` that is relative to the section. For example, `__sec_reduce_max_ind(x[40:10])` returns an index in the half-open interval $[0,10)$, *not* in $[40,50)$.

Two general reduction operations let you do reduction over a section with type T using your own **combiner** operation. Their signatures are:

```
T __sec_reduce(T initial, T section, T (*f)(T,T));
void __sec_reduce_mutating(T& dest, T section, U (*g(T*,T));
```

A summary of the arguments follows, with $\otimes$ denoting the combiner operation:

- *initial* is an initial value to use for the reduction.
- *section* is an array section.
- *f(x,y)* returns $x \otimes y$.
- *dest* is a location that contains the initial value for the reduction and is where the result is stored.
- *g(x,y)* sets $*x = *x \otimes y$. The function g can have any return type, because the return value of g is irrelevant.

Listing B.5 shows an example using string concatenation as the reduction operation.

```
#include <string>
#include <iostream>

using namespace std;

string concat( string x, string y ) {
   return x + " " + y;
}

int main() {
   string a[] = {"there","was","a","vector."};
   string b = __sec_reduce( "Once", a[0:4], concat );
   cout << b << endl;
   return 0;
}
```

LISTING B.5

Example of using `__sec_reduce` to reduce over string concatenation. It prints “`Once there was a vector.`”

B.8.4 Implicit Index

The built-in function `__sec_implicit_index(k)` returns the index of each element along dimension *k* in an array section implied by context. Examples include:

```
int a[5][8];
// Set a_{i,j} = i-j
a[0:5][0:8] = __sec_implicit_index(0)−__sec_implicit_index(1);
int b[8];
// Set b_{2k} = k
b[0:8:2] = __sec_implicit_index(0);
// Set b_{2k+1} = 10k
b[1:8:2] = 10*__sec_implicit_index(0);
```

The comments for the statements that set `b` show how the implicit indices are indices into the *section*, not the arrays in the expression.

B.8.5 Avoid Partial Overlap of Array Sections

In C and C++, the effect of a structure assignment `*p= *q` is undefined if `*p` and `*q` point to structures that partially overlap in memory. The assignment is well defined if `*p` and `*q` are either completely disjointed or are aliases for exactly the same structure. Cilk Plus extends this rule to array sections. Examples include:

```
extern float a[15];
a[0:4] = a[5:4];        // Okay, disjoint
a[0:5] = a[4:5];        // WRONG! Partial overlap
a[0:5] = a[0:5]+1:      // Okay, exact overlap
```

```
a[0:5:2] = a[1:5:2];        // Okay, disjoint, no locations shared
a[0:5:2] = a[1:5:3];        // WRONG! Partial overlap (both share a[4])
a[0:5] = a[5:5]+a[6:5];  // Okay, reads can partially overlap
```

The last example shows how partial overlap of reads is okay. It is partial overlap of a write with another read or write that is undefined.

Historical note: The original specification of array notation made partial overlap well defined, as in APL and Fortran 90. However, experience showed that doing so required a compiler to often generate temporary arrays, so it could fully evaluate the right side of a statement before doing an assignment. These temporary arrays hurt performance and caused unpredictable space consumption, both at odds with the C++ philosophy of providing abstractions with minimal performance penalty. So the specification was changed to match the rules for structure assignment in C/C++. Perhaps future compilers will offer to insert partial overlap checks into code for debugging.

B.9 #pragma simd

Analogously to how `cilk_for` gives permission to parallelize a loop, but does not require it, marking a `for` loop with `#pragma simd` similarily gives a compiler permission to execute a loop with vectorization. Usually this vectorization will be performed in small chunks whose size will depend on the vector width of the machine. For example, writing:

```
extern float a[];
#pragma simd
for ( int i=0; i<1000; ++i )
   a[i] = 2 * a[i+1];
```

grants the compiler permission to transform the code into something such as:

```
extern float a[];
for ( int i=0; i<1000; i+=4 ) {
   float tmp[4];
   tmp[0:4] = 2 * a[i+1:4];
   a[i:4] = tmp[0:4];
}
```

There is a subtle difference in the parallelization permited by `#pragma simd` versus `cilk_for`. The original loop in our example would *not* be legal to parallelize with `cilk_for`, because of the dependence between iterations. A `#pragma simd` is okay in the example because the chunked reads of locations still precede chunked writes of those locations. However, if the orginal loop body reversed the subscripts and assigned `a[i+1] = 2 * a[i]`, then the chunked loop would *not* preserve the original semantics, because each iteration needs the value of the previous iteration. In general, `#pragma simd` is legal on any loop for which `cilk_for` is legal, but not vice versa. In cases where only `#pragma simd` appears to be legal, study dependencies carefully to be sure that it is really legal.

A `#pragma simd` can be modified by additional clauses, which control chunk size or allow for some C/C++ programmers' fondness for bumping pointers or indices inside the loop. Note that

`#pragma simd` is not restricted to inner loops. For example, the following code grants the compiler permission to vectorize the *outer* loop:

```
#pragma simd
   for ( int i=1; i<1000000; ++i ) {
      while ( a[i]>1 )
         a[i] *= 0.5f;
   }
```

In theory, a compiler can vectorize the outer loop by using masking (Section 2.3) to emulate the control flow of the inner while loop. Whether a compiler actually does so depends on the implementation.

B.10 ELEMENTAL FUNCTIONS

An **elemental function** is a scalar function with markup that tells the compiler to generate extra versions of it optimized to evaluate multiple iterations in parallel. When you call an elemental function from a parallel context, the compiler can call the parallel version instead of the serial version, even if the function is defined in a different source file than the calling context.

The steps for using an elemental function are:

1. Write a function in scalar form using standard C/C++.
2. Add `__declspec(vector)`, and perhaps with optional control clauses, to the function declaration so that the compiler understands the intended parallel context(s) for calling it.[3] Additional clauses let you tell the compiler the expected nature of the parameters:
   `uniform(`*b*`)` indicates that parameter *b* will be the same for all invocations from a parallel loop.
   `linear(`*a*`:`*k*`)` indicates that parameter *a* will step by *k* in each successive invocation from the original serial loop. For example, `linear(p:4)` says that parameter `p` steps by 4 on each invocation. Omitting :*k* is the same as using `:1`.
3. Invoke the function from a loop marked with `#pragma simd` or with array section arguments.

Listings B.6 and B.7 show definition and use, respectively, of an elemental function. This code will likely perform better than a program where the function is not marked as elemental, particulary when the function is defined in a separate file. Writing in this manner exposes the opportunity explicitly

```
__declspec(vector(linear(a),uniform(b)))
void bar(float *a, float *b, int c, int d) {
   if( *a>0 )
      *a = b[c+d];
}
```

LISTING B.6

Defining an elemental function. The `declspec` tells the compiler to generate, in addition to the usual code, a specialized version for efficiently handling vectorized chunks where `a` has unit stride and `b` is invariant.

[3] Section B.10.1 describes `__attribute__` and C++11 attribute alternatives to `__declspec`.

```
__declspec(vector(linear(a),uniform(b)))
void bar(float *a, float *b, int c, int d);

void foo(float *a, float *b, int* c, int* d, int n) {
#pragma simd
   for( int i=0; i<n; ++i )
      bar( a+i, b, c[i], d[i] );
}
```

LISTING B.7

Calling an elemental function from a vectorizable loop. The `declspec` on the prototype tells the compiler that the specialized version from Listing B.6 exists. As usual in C/C++, a separate prototype is unnecessary if the function is defined first in the same file.

instead of hoping that a super optimizing compiler will discover the opportunity, which is particularly important in examples less trivial than this one.

Alternatively, the caller could call the elemental function from array notation, like this:

```
bar(&a[0:n], b, c[0:n], d[0:n]);
```

B.10.1 Attribute Syntax

If the compiler recognizes GNU-style attributes, you can use `__attribute__((vector))` to mark the function. As of this writing, the Intel compiler recognized both `__declspec` and `__attribute__` forms on Linux and Mac OS, but only the `__declspec` form on Windows.

Here are the first two lines of Listing B.6 written with the GNU-style attribute:

```
__attribute((vector(linear(a),uniform(b))))
void bar(float *a, float *b, int c, int d) {
```

Eventually elemental function markup will be expressible using C++11 attribute syntax.

B.11 NOTE ON C++11

C++11 **lambda expressions** (Section D.2) have a nifty application in Cilk Plus that in effect lets you spawn a statement. Here is an example that spawns the statement `while (foo())bar(baz);`:

```
cilk_spawn [&]{
   while (foo()) bar(baz);
}();
```

The code is really just spawning a functor created by the lambda. Do not forget the trailing parentheses—they are part of the spawned call. See also Section 14.6 for a novel use of a lambda expression with array notation.

B.12 NOTES ON SETUP

Cilk Plus does not require explicit setup in the user code; all capabilities self-initialize as needed. To compile code with Cilk Plus features, the inclusion of a number of Cilk Plus header files may be required. We recommend always using the header `<cilk/cilk.h>`, so you do not have to write "ugly" spellings like `_Cilk_spawn`. Other supplied header files are needed if you are using reducers or the API calls.

The serialization described in Section B.4 can be achieved by including `<cilk/cilk_stub.h>` or using the Intel compiler option `cilk—serialize`.

The Cilk Plus API, defined in `<cilk/cilk_api.h>`, provides some control over the runtime. By default, the number of worker threads is set to the number of cores on the host system. In most cases, the default value works well and should be used. However, you can increase or decrease the number of workers under program control using the following API call:

```
__cilkrts_set_param("nworkers",n)
```

or via the environment variable `CILK_NWORKERS`. You may want to use fewer workers than the number of processor cores available in order to run tests or to reserve resources for other programs. In some cases, you may want to oversubscribe by creating more workers than the number of available processor cores. This may be useful if you have workers waiting on locks, or if you want to test a parallel program on a single-core computer.

There are additional detailed "under the hood" controls and information exposed in the Cilk Plus API—for example:

`__cilkrts_get_worker_number()`	Gets worker number.
`__cilkrts_get_total_workers()`	Gets total number of workers.

Consult the Cilk Plus documentation for more information. We stress that using the two queries as part of an algorithm is usually a sign of bad style in Cilk Plus. The whole point of Cilk Plus is to abstract away the number of workers.

B.13 HISTORY

The Cilk language has been developed since 1994 at the MIT Laboratory for Computer Science. It is based on ANSI C, with the addition of just a handful of Cilk-specific keywords.

Cilk is a faithful extension of C and the serial elision (Section B.4) of any well-defined Cilk program is always a valid serial implementation in C that matches the semantics of the parallel Cilk program. Despite several similarities, Cilk is not directly related to AT&T Bell Labs' Concurrent C.

In the original MIT Cilk implementation, the first Cilk keyword was `cilk`, which identified a function as written in Cilk. This keyword was needed to distinguish Cilk code from C code, because in the original implementation Cilk procedures could call C procedures directly, but C procedures could not directly call or spawn Cilk procedures.

A commercial version of Cilk, called Cilk++, that supported C++ and was compatible with both GCC and Microsoft C++ compilers was developed by Cilk Arts, Inc. The `cilk` keyword morphed into `extern "cilk"`. Cilk++ introduced the notion of hyperobjects [FHLLB09], which elegantly eliminated the need for several keywords in the original Cilk pertaining to reductions.

In July 2009, Intel Corporation acquired, from Cilk Arts, the Cilk++ technology and the Cilk trademark. In 2010, Intel released a commercial implementation in its compilers combined with some data parallel constructs, under the name Intel Cilk Plus. Intel has also released specifications, libraries, code, and the ability to use the "Cilk Plus" name (trademark) with other compilers.

Intel Cilk Plus extends Cilk and Cilk++ by adding array extensions, being incorporated in a commercial compiler (from Intel), and having compatibility with existing debuggers. Intel Cilk Plus adopted a significant simplication proposed by Cilk++ team: Eliminate the need to distingush Cilk linkage from C/C++ linkage. This was a major improvement in usability, particularly for highly templated libraries, where linkage specifications can become confusing or impossible. Furthermore, erasing the distinction between Cilk and C/C++ functions enabled C/C++ functions to be spawned directly.

Intel has published both a language specification and an ABI specification to enable other compilers to implement Cilk Plus in a compatible way and to optionally utilize the Intel runtime. The Cilk Plus extensions to C and C++ have also been implemented in a development branch version of the GCC compiler.

Intel has stated its desire to refine Cilk Plus and to enable it to be implemented by other compilers to gain industry-wide adoption.

B.14 SUMMARY

Cilk Plus is a language specification that provides for both thread and vector parallelism in C and C++ via keywords, syntax for array operations, elemental functions, and pragmas. Much more information is available at `http://cilkplus.org`.