Fermi National Accelerator Laboratory

FERMILAB-TM-1881

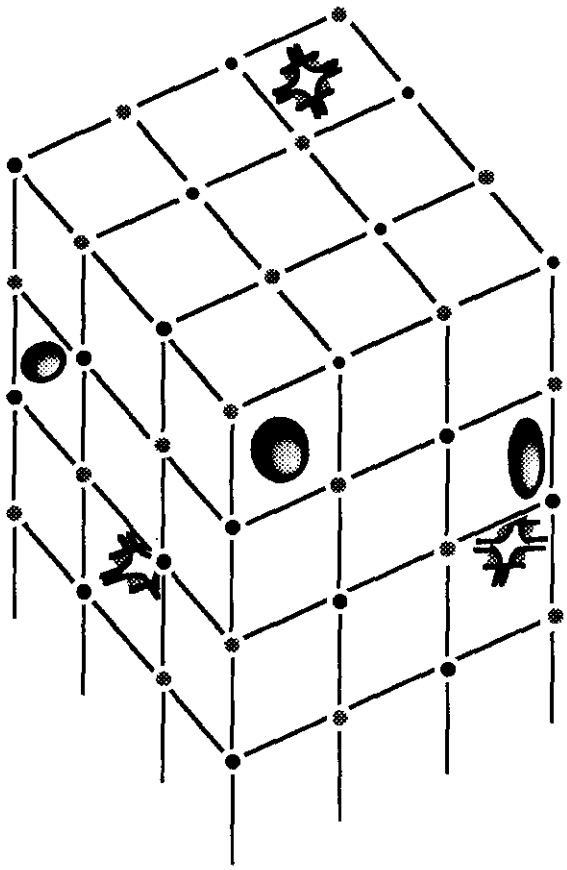
Canopy Version 7.0: Canopy Manual

Mark Fischler, Mike Uchima, George Hockney and Paul Mackenzie

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

March 1994

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy



Canopy 7.0 • December 1993

Fischler • Hockney
Mackenzie • Uchima

Fermi National Accelerator Laboratory

Canopy Manual

Fermi National Accelerator Laboratory

Canopy Version 7.0: Canopy Manual

Mark Fischler and Mike Uchima
Fermilab Computer R&D Department
and
George Hockney and Paul Mackenzie
Fermilab Theoretical Physics Group

Contents

Chapter 1

Introduction

Canopy provides a machine-independent environment for attacking grid-oriented problems. This document describes the concepts and routines common to all Canopy implementations, independent of the system and implementation. Information specific to the massively parallel ACPMAPS/indexACPMAPS system at FermiLab is contained in two other documents: The CANOPY ACPMAPS USER'S GUIDE provides user-oriented instructions on compiling, running, file system usage, and production job control. The CANOPY ACPMAPS SYSTEM MANUAL describes system tools and installation and system management techniques. System-specific User's Guides may be created for implementations on other systems.

The goal of Canopy is to allow scientists to use massively parallel systems for a broad class of applications without having to become expert in any particular system or in parallel programming techniques. The Canopy approach identifies grid-oriented concepts and implements them as routines in a library. Applications written in terms of these concepts will run on any system which supports the Canopy software. A side benefit in dealing with familiar concepts is that programs can more easily be understood by other researchers.

1.1 Manual Organization

The INTRODUCTION and the second and third chapters in this manual, CONCEPTS IN CANOPY and A TUTORIAL EXAMPLE, familiarize the user with with the Canopy paradigm. They assume, however, that the user already knows something about both C and numerical analysis. Together they give an overview of the way Canopy programs work.

The fourth chapter, PARALLELISM CONSIDERATIONS, describes consequences of the assumption that Canopy programs run on parallel, distributed memory systems.

The fifth and sixth chapters, STRUCTURES AND TYPEDEFS and CANOPY SUBROUTINE REFERENCE, form a reference manual for the Canopy software. They describe the provided subroutines (and user-provided customization functions) and explain how to use them.

The seventh chapter describes standard Canopy libraries. GRIDLIB is a library of functions to define different sorts of Canopy grids. Currently it contains only periodic rectilinear grids but these subroutines may be used as examples to define grids with other topologies. SETLIB is a library of common set functions which are mostly related to rectilinear grids. RANLIB is a library of random number generators suitable for the massively parallel environment. Currently, it contains routines only for uniformly-distributed randoms. CMPLXLIB is a library of complex number macros, typedefs, and subroutines. FFTLIB is a library of FFT routines for periodic rectilinear grids. It contains fast multi-dimensional FFT's for arrays whose sizes are multiples of three and five as well as factors of two. PROMPTLIB is a library of prompted input routines which recover from errors in the input and handle terminal and re-directed file I/O appropriately.

As a general rule, references to sections of the manual are set in SMALL CAPS and references to C or Canopy objects are set in `typewriter face`.

1.2 Perspectives on Usage

Canopy is a library of C-linkable subroutines designed to guide the organization of parallel software. Properly written programs using these subroutines will automatically take advantage of parallel hardware. However, it makes perfect sense to compile, link and run Canopy programs on a single-node machine as well, and is often useful to do so while debugging. A Canopy application can be run on any system which supports the Canopy software (any "Canopy platform"). Care has been taken to give platform-independent results.

Canopy applications are structured as a *control program* which issues *tasks*. The strategy of using a parallel system is that the exectution of these tasks is distributed over many *nodes*. A node consists of a CPU along with its own memory; at any time (including during a task), a node may access its own memory or the memory of other nodes. Canopy routines facilitate the access to data which may or may not reside on the local node. (See Section 2.3 CANOPY PROGRAM ORGANIZATION.)

As Canopy is built on C, almost all C programming concepts apply. In particular, since C is built on UNIX, system calls and program invocation follow UNIX conventions. Canopy is generally written to use POSIX system calls where-ever possible to enhance portability. Therefore Canopy programs look almost like UNIX C programs.

However, there are a few subtle differences arising from the parallel nature of Canopy. These are gathered together here and apply to all Canopy platforms

> Canopy programs have `control()` instead of `main()` as their main entry point. Just as `main` returns an exit code and has `argc` and `argv` and `envp` arguments, so does `control`. Using `control` this way allows Canopy set-up to be done before the program starts.

> Canopy modules are compiled and linked with a special tool instead of `cc` which defines appropriate macros and links appropriate libraries. The `canc` tool prepares modules

for the native UNIX machine. Tools for module preparation on specific massively parallel systems are described in system-specific user's guides. For example, the dcanc and acanc tools prepare modules for the multi-node ACPMAPS machine; these are detailed in the CANOPY ACPMAPS USER'S GUIDE. On these tools, most of the cc flags work in the normal way.

All Canopy programs should call `complete_definitions()` once and only once. This is required to set up the grids and fields but also ensures the linker will find the `main` program. Sometimes a program without this call will link correctly but it is a good idea to include it.

I/O should not be done inside task routines. This is mostly a logical restriction, since the order of I/O would then be undefined. Sometimes it is useful to print from inside a task and while this is allowed it is discouraged except for debugging. Input is not supported. The result of output is not completely defined and may cause surprises, even if `fflush` is used.

The use of global variables is discouraged since the compiler cannot tell if their use is legal. In general they should only be used to pass data between different parts of the control program or different parts of the task routine on the `same` site or to pass data `from` the control program `to` all the task routines using `broadcast`. Other uses are probably not logically correct in ways that are hard to find. Sending data through `do_task` is much safer.

The `setjmp` and `longjmp` routines are not allowed. While these may work in some implementations (and do strange things) they are not required of a valid Canopy platform. In particular, `setjmp` and `longjmp` must not jump into or out of tasks.

Canopy modules created with canc work exactly the same way normal C programs do, picking up the environment and command line arguments the same way. They are invoked simply by typing their names and work with pipes in the normal UNIX fashion. Multi-node Canopy modules also pick up environment and command line arguments and work with pipes, but using them requires understanding the ACPMAPS system tools. Notice that, as with all cross-compiler tools, `.o` object files created with different compilers are incompatible and therefore must be kept separate.

1.2.1 Canopy on UNIX

On UNIX systems, the Canopy-specific tool canc is used instead of cc to compile and link Canopy programs. Here is a simple Canopy program which does not make use of the Canopy grid or parallelism concepts:

```
/*                                                    */
/* file hi.c, almost the simplest Canopy program */
/*                                                    */
#include <canopy.h>

void control(int argc, char **argv, char **envp)
                          /* pick up command line */
{
  int i;
  printf("Hi Guys!\n");
  complete_definitions();      /*significant for linker*/
  printf("argc = %d\n",argc);
  for (i=0; i<argc; i++) {
    printf("argv[%d] = %s\n",i,argv[i]);
  }
} /* control */
```

canc works like cc, in particular obeying the command line options `-c`, `-o <file>`, `-D<name>`, and `-l<lib>`. To use canc to compile the

example do this:

```
>canc hi.c -o hi
```

The hi module is an ordinary UNIX module which runs in the ordinary UNIX way:

```
>hi users
Hi Guys!
argc = 2
argv[0] = hi
argv[1] = users
```

1.3 Perspectives on Implementation

Although Canopy is designed as an environment providing the concepts natural to grid-oriented applications, it is also designed for efficient implementation on a variety of massively parallel, distributed memory systems. The Canopy software implementation is organized with that in mind, striving for ease in portability. There are two aspects to portability: Applications making use of Canopy routines should run on any Canopy platform; and it should be straightforward to port the Canopy software to suitable new systems.

To achieve these goals, Canopy is designed as a layered product, with each layer presenting a clean interface to the higher layers. The lowest layer, the Canopy Hardware Interface Package (or CHIP), provides routines unifying different machines. The public interface to the CHIP routines—and all routines in layers above CHIP—will be machine-independent. The next layer, Canopy, provides grid and task primitives, implementing the concepts needed for grid-oriented applications. On top of that are libraries built on the Canopy routines (and using the public CHIP interfaces where necessary): The set, grid, prompt, cmplx, and fft libraries provide specific tools within the Canopy framework. The highest layer is, of course, the user application. Each layer has access to *all* of the public concepts of the lower layers.

The CHIP layer is designed to isolate the higher layers from machine details. Intended as a base for Canopy and other higher-level application packages, it introduces concepts for dealing with single- or multi-node machines in a unified way, so that details such as the exact flavor of C or the number of nodes are unimportant above this layer. All the mechanics of synchronizing and controlling parallel CPU's are provided by CHIP, including: an overall paradigm for system addresses; routines to read and write to those addresses; the general job start-up and running procedures; fatal error handling; semaphores; and a facility to run subroutines on all nodes.

The Canopy layer introduces concepts appropriate for solving grid-oriented problems, including grids, sites, sets, directions, paths, fields, maps, and tasks. It contains a large number of subroutines for manipulating the concepts discussed in the second chapter of this manual. The CANOPY SUBROUTINE REFERENCE chapter of this manual describes the public Canopy and CHIP subroutines in detail, and explains how to use them. The last section of this chapter describes the public CHIP routines,

Chapter 2

Concepts in Canopy

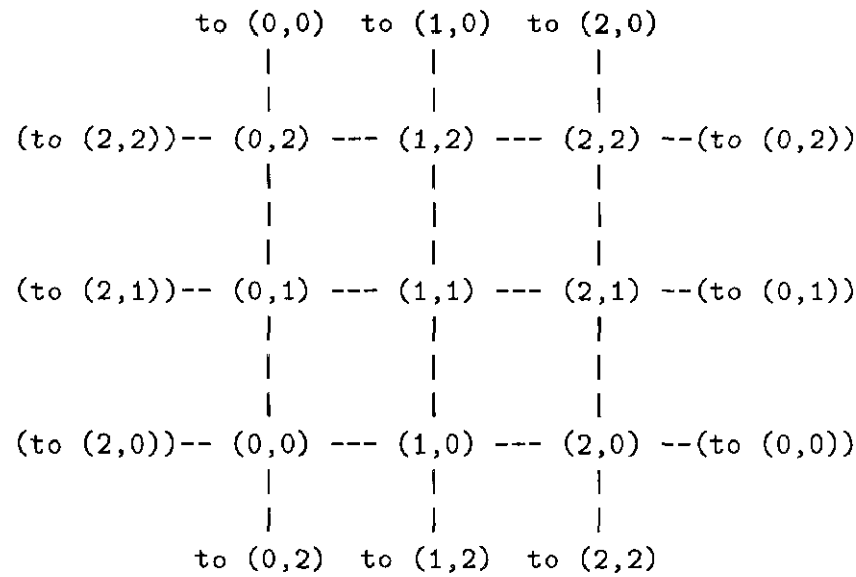
Canopy provides a natural environment for solving grid-oriented problems. It includes concepts such as links, sets, and fields over grids; and subroutines for doing numerical analysis operations that deal with these concepts. Since it is designed to run in a parallel environment (the massively parallel ACPMAPS system), it uses the natural parallelism in most grid-oriented problems automatically, isolating the user from the multi-node nature of the machine. By using the grid-oriented concept of performing the same operation on many sites of the grid simultaneously, it encourages more structured programming as well as efficient use of the parallel processing machine.

2.1 Grid-Oriented Structures

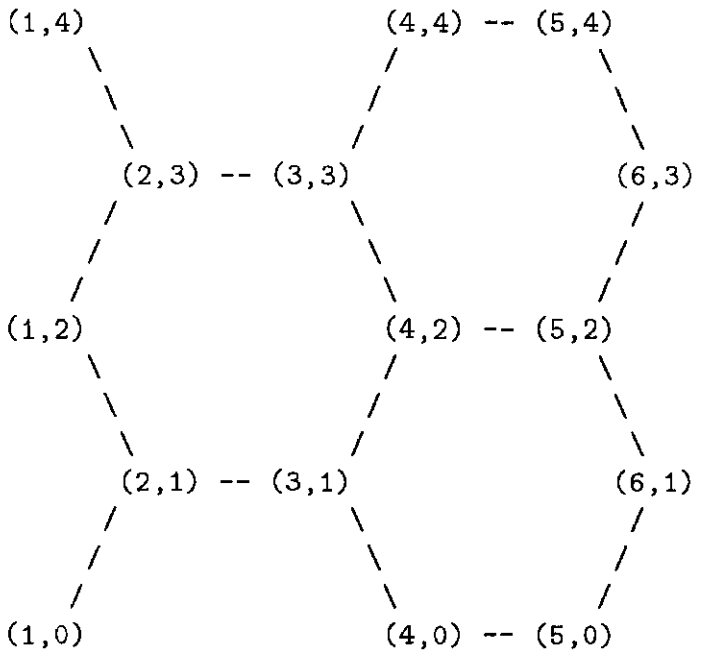
The underlying arena on which grid-oriented problems are solved is a *grid* consisting of *sites* (or points on the grid) and *links* connecting these sites. Canopy requires the additional concepts of the *coordinates* of a site and the *direction* of a link. Each site is referenced by its coordinates, and each link by its direction from some site.

Each link may therefore be named in two ways since it connects two sites. The link in direction $+d$ from site A to site $A+d$ must be the link in direction $-d$ from site $A+d$ to site A. It is not trivial to require links

and directions to be labeled this way, although it is natural to do so on rectangular grids. For lattice gauge theory programs in four dimensions on a regular hypercubic grid there are four positive directions `X`, `Y`, `Z`, and `T`; and the four corresponding negative directions. On a two-dimensional rectangular grid there are two positive and two negative directions, and the labelling looks as follows:



For other types of grids the `direction` and `coordinate` structure can be more complicated. For example, one way to set up a hexagonal grid is to use *three* positive directions on the plane: `+x000`, `+x120` and `+x240`; and the corresponding negative directions `-x000`, `-x120`, and `-x240`. Then, as is shown in the diagram on the next page, some sites only have links in the positive directions and some have links only in the negative directions, so not all directions lead to links from every site. The coordinate assignment is also arbitrary, and some coordinates, such as `(1,1)` in the diagram, do not correspond to sites.



The concepts of `coordinate` and `direction` do not always have direct physical meaning. For example, a problem done on a mesh model of a surface where each site always has four nearest neighbors can easily be handled in Canopy by mapping the surface to a two-dimensional rectangle with appropriate boundary conditions and using the `X` and `Y` directions. It is also possible to define irregular grids by increasing the number of directions, although the unused directions will take up memory space.

The standard library `gridlib` contains definition routines for periodic rectilinear grids. Other grids, such as the example hexagonal grid, may be defined by the user with the Canopy subroutine `arbitrary_grid`.

2.1.1 Paths

A path is a list of directions showing how to move from site to site along the links of a grid. Paths are a powerful concept because they can express algorithms in a position-independent way. For example, a common object in lattice gauge theory is the product of link matrices around a *plaquette* or square formed by the path *+d1*, *+d2*, *-d1*, *-d2*. The plaquette can be calculated from any site on the grid without reference to coordinates, even if the path crosses a boundary; all that matters is the grid connectivity set up at the beginning of the Canopy program.

However, not all paths need be valid from all sites since not all sites have links in all directions. On the hexagonal grid example there are really two types of sites (those with links in the `+x000`, `+x120`, and `+x240` directions and those with links in the `-x000`, `-x120`, and `-x240` directions), and paths valid for one would not be valid for the other. It is also possible to define a rectilinear grid without periodic boundary conditions where the sites on the edge have no neighbor in the direction of the edge. In this situation paths are still useful but some of them may lead `NOWHERE` and the program must account for this.

2.1.2 Fields

Many physical problems involve *fields* of some sort defined on the continuum, such as the electric field on space or the gluon field on spacetime. When problems involving fields are solved by a grid approximation, the continuum fields are replaced by fields on the sites or links of the grid. Canopy allows `fields` to be defined which reserve space for *field elements* on either the sites or links of a grid and provides a set of subroutines for manipulating the field elements. Note the distinction between a field element, which is some structure stored on a `site`, and a `field`, which is a variable used to refer to a particular group of structures on sites. The `field` concept is one of the ways Canopy hides the multi-node nature of the machine. Field elements are distributed over the nodes automatically, but the user always references field ele-

ments through Canopy subroutines so the actual distribution of field elements to nodes never affects the high-level user programs. Even on a single-node machine this construct improves modularity by making all references to field elements (which are a kind of global data) only through conceptually simple subroutines.

2.1.3 Sets

After `grids` and `fields` have been defined Canopy is ready to do operations on some `set` of sites on the grid, such as all sites on a grid or the sites on the grid boundary. The `do_task` subroutine loops over all the sites in a `set`, calling some *task routine* on each one. These calls to the task routine are automatically distributed over nodes and are done in parallel. The task routine does something relative to its `HOME` site (the current site in the `do_task` loop). Canopy does parallel operations only through calls to `do_task()` on some `set` of sites.

2.1.4 Maps

A `map` establishes relations between grids. Each site on a *domain* grid is mapped either to some site on a *range* grid or to a special site called `NOWHERE`. These maps may be one-to-one or many-to-one and need not cover the entire *range* grid. Canopy provides subroutines for finding the image or inverse image of a site, and for doing operations on each site in the inverse image of a site.

Maps are intended for problems where there is an obvious sub-grid structure. For example, it is sometimes useful to sum up all elements of one time-slice of a four-dimensional grid. By creating a separate time grid and mapping each element of the four-dimensional grid to its time coordinate in the time grid, a task can be written for the time grid which does a sub-task to sum up the elements of the four-dimensional grid for each time slice. This allows all the time slices to be summed up in parallel, which is hard to do any other way.

2.2 Canopy Variables

To implement the above concepts Canopy uses the following variable types which are all defined in C `typedef` statements:

`grid`: Variables of type `grid` identify particular grids.

`set`: Variables of type `set` identify particular sets.

`map`: Variables of type `map` identify particular maps.

`field`: Variables of type `field` identify particular fields. These are used by Canopy subroutines with `site` and `direction` variables to fetch and store *field elements*, which may be any C structure.

`site`: Variables of type `site` identify particular sites on a grid. There is a special variable inside task routines, `HOME`, which points to the current site in the loop over sites done by the `do_task` routine. Another special site, `NOWHERE`, is used by map routines to indicate that the image or inverse image of a map is empty.

`direction`: Variables to indicate particular directions, which range from `-ndir` to `+ndir` (skipping 0). For example, `grid.h` defines the symbols `MINUS_T`, `MINUS_Z`, `MINUS_Y`, `MINUS_X`, `X`, `Y`, `Z` and `T` as the numbers −4 to 4, appropriate for hyper-cubic grids.

`coordinates`: This is a 1-based array of integers used to specify a site's coordinates. For the above-mentioned hyper-cubic grid the directions `X`, `Y`, `Z` and `T` are indices for this array.

`path`: This type is a pointer to a list of directions.

These types are described in detail in the TYPEDEFS and CANOPY SUBROUTINE REFERENCE sections. There are some additional types, used mainly for internal reference and optimization, but they are built around the same concepts.

2.3 Canopy Program Organization

Canopy programs are divided into three parts: the declaration section where grids, fields, sets, and maps are defined; the single-thread executable section which does overall process control and calls parallel tasks; and tasks, which do some operation in parallel on a set of sites. The first and second sections together compose the *control program*, so called because its main entry point must be named `control()`.

2.3.1 Control Program Declaration Section

The declaration section is the place where all the grids, fields, sets and maps are declared. This section can read input from a file or terminal and then decide what sizes to use, so very flexible programs may be written. After completing the declaration section the program must call `complete_definitions`, which reserves space for and sets up pointers to the declared structures.

2.3.2 Control Program Executable Section

The executable section of the `control` program actualy executes the algorithm. It can invoke `do_task()` to call the task routines, which are executed in parallel. Both the declaration and the executable section of the `control` program may call subroutines which are still considered part of the `control` program for all system purposes.

2.3.3 Task Routines

Task routines do the same thing on each site for which they are called. Typically they use the `HOME` site, which is set by the `do_task()` call to each site in the set in turn, as a base site. Some fields are gathered in directions or paths relative to `HOME`, some calculation is done and the results stored back in a field on the `HOME` site.

The task routines must be written so that each site in the set may be acted on without interfering with the task routine on other sites

of the same set. Thus Canopy can run on all of the sites in parallel without any additional user intervention. Provisions are made for *compound* sets of sites, where do_task must call the task routine in some specific order. The PARALLELISM CONSIDERATIONS chapter describes the special subroutines needed to manage this situation properly.

2.4 Summary

Canopy is a framework for writing programs that solve grid-oriented problems. Subroutines to create grids with known connectivity are built into it, as are subroutines to manipulate the fields that live on these grids. Parallelism is automatically achieved by writing task routines which do an operation on a site. The user need not be aware of machine details such as the number of nodes or the distribution of sites to nodes. With Canopy, programs can be debugged on a normal single-thread machine and then moved to a multi-node system without change.

Chapter 3

A Tutorial Example

Perhaps the simplest and best-known grid-oriented problem is the Laplace heat equation, $\nabla^2\phi = 0$. It can be solved numerically by a relaxation algorithm. The program `laplace.c`, which solves this equation on a rectangular grid, is a good example of how Canopy functions fit together. This chapter discusses `laplace.c` in detail and also explains how to use other Canopy concepts to solve more complicated problems. The basic organization of the example program will be shared by all Canopy programs.

To solve $\nabla^2\phi = 0$ on a rectangular grid, start by writing the differential equation as a difference equation:

$$\nabla^2\phi = 0 \rightarrow ((\Delta^2)_x + (\Delta^2)_y)\phi = 0 \tag{3.1}$$

or

$$((\phi_{\mathbf{x}+\hat{x}} - \phi_{\mathbf{x}}) - (\phi_{\mathbf{x}} - \phi_{\mathbf{x}-\hat{x}})) + ((\phi_{\mathbf{x}+\hat{y}} - \phi_{\mathbf{x}}) - (\phi_{\mathbf{x}} - \phi_{\mathbf{x}-\hat{y}})) = 0 \tag{3.2}$$

which yields the equation

$$\phi_{\mathbf{x}} = \frac{\phi_{\mathbf{x}+\hat{x}} + \phi_{\mathbf{x}-\hat{x}} + \phi_{\mathbf{x}+\hat{y}} + \phi_{\mathbf{x}-\hat{y}}}{4}. \tag{3.3}$$

This may be solved by a relaxation algorithm which replaces the value at each point on the grid with the average value of its nearest neighbors.

The program `laplace.c` (there is a copy of it available in Canopy source files) implements this algorithm. Lines from `laplace.c` appear roughly in order throughout this chapter with explanations. Its main sections work as follows: First, it sets the boundary conditions on the grid, which are fixed throughout all the relaxation sweeps. This is actually one of the hardest parts of the program to write since considerable thought must be given to various details. In `laplace.c`, the program takes as input the values of the temperature at the four corners of the rectangle and then interpolates these values along the edge, thereby eliminating any possible discontinuity problems. Then the program sweeps over the interior sites iteratively, replacing the temperature at each site with the average of the temperature of its four neighbors. The sites are updated in a red/black order and the new values of the red sites are used to update the black sites. For the convergence of the Laplace equation this ordering is unimportant, but for other problems it is crucial. The PARALLELISM CONSIDERATIONS chapter discusses some of the subtler issues involving the order. Finally, the program uses a subroutine to print out the initial and final temperature values in a user-readable form.

As usual in a C program, lower-level routines precede the top routines which call them. In the context of this Canopy TUTORIAL example, that means the overall control program appears at the end; the individual task routines (to be executed in parallel) are presented before that; and the routines defining sets of sites over which the tasks are to work are presented first.

3.1 Canopy C Conventions

Canopy is designed to run under C compilers which conform to the ANSI standard. Canopy applications should be written in ANSI C style.

Since Canopy existed before the ANSI C standard, and the compiler for the Weitek processor used in the original ACPMAPS system was not ANSI C, the Canopy compilation tool applies a pre-processor

which performs necessary conversions such as including changing ANSI prototypes and function declarations to the old Kernighan and Ritchie style.

In order to be compatible with the preprocessor, programs should use the `voidptr` type instead of `void*` and should not use array types in function argument lists (that is, avoid constructs such as `int f(int x[3])`)—use `typedefs` instead. Also, variable length argument lists pose a special problem since there are many ways to handle them. The pre-processor does not attempt to convert these automatically. However, the Canopy routines which take variable length argument lists (`do_task` and `make_path`) are declared in `<canopy.h>` in the correct way for several target systems.

3.2 Required Include Files

Every Canopy program must include the `<canopy.h>` file, which contains function prototypes and definitions for all the Canopy and CHIP functions. In additions to `<canopy.h>`, the Laplace program uses routines from `<math.h>`, `<prompt.h>`, and `<grid.h>`. The math library is standard C and the prompt and grid libraries is described in this manual. `MINUS_Y`, `MINUS_X`, `X` and `Y` are `#defined` in `<grid.h>`.

```
#include <canopy.h>
#include <grid.h>
#include <math.h>
#include <prompt.h>
```

3.3 Grid and Set Defining Functions

A Canopy program may contain functions defining one or more grids, and functions to define sets of sites on the grids. The example Laplace program will use `periodic_square_grid()` defined in `<grid.h>`, so no grid definition functions appear here. The program defines five simple and one compound set of site functions, some of which are used as building blocks for the others. Some of these set functions are also in

`setlib`, but are repeated here for clarity. If `setlib` functions are used, then `set.h` should also be included.

The `red_func` function sets up a simple set of sites whose sum of coordinates is odd. It does this by returning `1` for those sites with odd coordinate sums and `0` for those sites with even coordinate sums. This function is not actually used except to build the other functions, but it could be used to define a set of sites. Note how the number of dimensions is obtained from the `lattice` argument:

```
int red_func(grid lattice, intptr coords)
    /* red sites have odd coordinate sums */
{
  int ndim = number_of_dimensions_of_grid(lattice);
  int i;
  int csum=0; /* note that coords is 1-based! */
  for (i=1; i<=ndim; i++) { csum += coords[i]; }
  return(csum % 2); /* 1 for red; 0 for black */
} /* red_func */
```

The `black_func` function sets up a simple set of sites whose coordinate sum is even by returning `1` if the coordinate sum is even and `0` if it is odd. It uses `red_func` to do the work. This function also is not used except to build other functions:

```
int black_func(grid lattice, intptr coords)
    /* black sites have even coordinate sums */
{
  return (1 - red_func(lattice,coords));
} /* black_func */
```

The `boundary_func` function sets up a simple set of sites which lie on the edge of the grid. This function actually is used to make a set of sites, since the initialization is done only on the boundary sites and the update only on the interior sites. It uses `grid_upper_bounds()` and `grid_lower_bounds()` to find out the grid dimensions without explicitly using any global variables.

```
int boundary_func(grid lattice, intptr coords)*/
    /* sites on the boundary -- using Canopy data */
{
  int ndim = number_of_dimensions_of_grid(lattice);
  int *max = grid_upper_bounds(lattice);
  int *min = grid_lower_bounds(lattice);
  int i;
              /* coords, min, and max are 1-based */
  for (i=1; i<=ndim; i++) {
    if ((coords[i]==min[i])||(coords[i]==max[i])) {
      return(1);/* boundary sites are in the set  */
    }
  }
  return (0);              /* other sites are not */
}
```

Now the above three functions are used to make functions for the red interior sites and the black interior sites, which are the sets used for the Laplace update. As was mentioned above, the Laplace algorithm converges no matter what order is chosen for the field update. The red/black order is chosen due to subtle issues involving parallism that are described in detail in the PARALLELISM CONSIDERATIONS chapter, which gives rules on how to choose acceptable orderings.

```
int red_interior_func(grid lattice, intptr coords)
    /* red sites not on the boundary */
{
   int i;
   if (boundary_func(lattice,coords) == 1) {
     return(0); /* boundary sites are not in set  */
   }
   /* otherwise red sites are 1 and black sites 0 */
   return (red_func(lattice, coords));
}
```

```
int black_interior_func(grid lattice, intptr coords)
    /* black sites not on the boundary */
{
  int i;
  if (boundary_func(lattice,coords) == 1) {
     return(0); /* boundary sites are not in set  */
  }
   /* otherwise red sites are 0 and black sites 1 */
  return (black_func(lattice, coords));
}
```

Finally, `rb_interior` is the compound set of sites. It sets up the set of all interior sites in a red-then-black order. For boundary sites it returns 0, for interior red sites 1, and for interior black sites 2. This compound set of sites could be used for updating red sites first and then black sites, if appropriate changes are made to the task routines. Again there is more discussion of these techniques and their implications in the PARALLELISM CONSIDERATIONS chapter.

```
int rb_interior(grid lattice, intptr coords)
    /* sites not on  the boundary, in red-black order */
{
 int i;
   if (boundary_func(lattice,coords) == 1) {
     return(0); /* boundary sites are not in set  */
   }
   /* otherwise red sites are 1 and black sites 2 */
 return (1 + black_func(lattice, coords));
}
```

3.4 Task Routines

Task routines are the heart of Canopy programming. They do some operation on the `HOME` site, such as initializing or updating one of its fields using the Canopy subroutines. Task routines are called by the `do_task` subroutine once for each site in some set, with the `HOME` site taking on the value of each site in the set. `do_task()` automatically distributes the calls to the task routine over all the available nodes so

operations may be done in parallel. A typical do_task call looks like this:

```
float f;
float x[4];
do_task(task_routine, set_to_do,
        PASS, &f, sizeof(f),
        INTEGRATE, &x, 4*sizeof(float),
        END);
```

This causes the subroutine task_routine to be called once for each site in the set set_to_do. The remaining arguments of do_task() describe the arguments of task_routine and what is to be done with them. In this example, task_routine would look like this:

```
void task_routine(float *farg, float *xarg)
{
  /* code where *farg is read-only but   */
  /* four elements of xarg[] are written */
} /* task_routine */
```

The two arguments of task_routine are farg, which is used as a pointer to a float, and xarg which is used as a four-element floating point array. The array is filled with four values. The argument triplets in the do_task call tell the system exactly how to treat arguments to task_routine (see Section 4.3, ARGUMENTS TO TASK ROUTINES). The (PASS, &f, sizeof(f)) triplet says that the first argument to task_routine is *read-only*, that its address is &f, and that it has the size of a float. The second triplet says that the second argument of task_routine is to be *summed up* over all the sites in the set as a *floating-point array* with four elements. The array at &x will be filled with the answer—note that do_task initializes it to zero first. The last "triplet," END, says there are no more arguments. This complicated structure is needed in order to properly call task_routine on many different processors simultaneously and keep track of the results.

The task routines in laplace.c use a "generalized subroutine header" structure which lets the calling routine call what looks like an ordinary C subroutine that handles the do_task call internally. There are

two reasons for using this: 1) the `control` program will look simpler and 2) the error-prone ugliness of passing input scalars by address instead of by value is eliminated. The "generalized subroutine header" format requires two routines with almost the same name: one of these is the actual task routine that appears in the `do_task` call and the other is a header routine that calls `do_task`. The first routine has the same name as the second with an underscore character appended. In the first example below, the task routine is `set_field_to_zero_` and the header routine is `set_field_to_zero`.

The header routine has one more argument than the task routine, since the header routine needs to know which set to use in the `do_task()` call. Conventionally, this set is the first argument of the header routine. The remaining arguments of the header routine are the same as the arguments of the task routine except that scalar input arguments such as `fields`, `maps`, `grids`, `sets`, `ints` and `floats` may be pass-by-value in the header routine. All arguments of the task routine are pass-by-address, since `do_task` uses the address of the argument.

The first example is a simple task routine which sets a field of floats to zero on its `HOME` site. It has only one argument: the field to be set to zero. This is passed by value to the header routine and passed by address in the task routine. This routine is rather specialized since it needs to know both that the field is a field of floats and that the length of each field element is one. It would be possible to write a more general routine that uses the `field_length()` function to zero a field of arbitrary length but for `laplace.c` this is not needed.

```
                /* first the actual task routine,  */
                /* complete with ending underscore */
void set_field_to_zero_(field *f)
    /* zero HOME site of f */
/* in task routines, always pass arguments by address */
{
      float zero = 0.0;
      put_field(*f, HOME, (voidptr) &zero);
}
```

```
                /* now the header routine with the */
                /* same name but no underscore     */
void set_field_to_zero(set set_of_sites, field f)
/* note that here arguments can be pass-by-value */
{
  do_task(set_field_to_zero_, set_of_sites,
          PASS, &f, sizeof(field),
          END);
} /* set_field_to_zero */
```

The second task routine does a relaxation sweep on the HOME site. It replaces the field at the HOME site with the average of the field at the nearest neighbors. The header structure is exactly the same as before.

```
void laplace_sweep_(field *f)
/* in task routines, arguments are pass-by-address */
{
  float a =
  *(float*)field_pointer_at_dir(*f, X) +
  *(float*)field_pointer_at_dir(*f, Y) +
  *(float*)field_pointer_at_dir(*f, MINUS_X) +
  *(float*)field_pointer_at_dir(*f, MINUS_Y);
  a /= 4.0;
  put_field (*f, HOME, (voidptr) &a);
}
                /* now the header routine with the */
                /* same name but no underscore     */
void laplace_sweep(set set_of_sites, field f)
/* note that f is pass-by-value here */
{
  do_task(laplace_sweep_, set_of_sites,
          PASS, &f, sizeof(field),
          END);
} /* laplace_sweep */
```

The third task routine is called at the beginning of the program to initialize all the sites on the boundary. It uses the grid information routines to get the upper and lower coordinate limits and is passed an array of floats containing the values in the corners, which it uses to interpolate boundary values for the edges. Since it only makes sense

to call this task with the boundary set of sites it would be possible to re-write the header to use the boundary set directly. This would require that the boundary set be a global variable instead of local to the main program, however.

```
void set_boundary_sites_(float *corner_value, field *f)
{
  grid lattice = grid_supporting_site(HOME);
  int *max = grid_upper_bounds(lattice);
  int *min = grid_lower_bounds(lattice);
  coordinates coords;    /* temp coordinate array */
  float val;

  get_coordinates (HOME, coords);
  if (coords[X]==min[X]) {              /* top */
    val = (corner_value[1] - corner_value[0]);
    val *= coords[Y] - min[Y];
    val /= max[Y]-min[Y]+1;
    val += corner_value[0];
    put_field (*f, HOME, (voidptr) &val);
  } else if (coords[X]==max[X]) {       /* bottom */
    val = (corner_value[2] - corner_value[3]);
    val *= coords[Y]-min[Y];
    val /= max[Y]-min[Y]+1;
    val += corner_value[3];
    put_field (*f, HOME, (voidptr) &val);
  } else if (coords[Y]==min[Y]) {      /* left */
    val = (corner_value[3] - corner_value[0]);
    val *= coords[X]-min[X];
    val /= max[X]-min[X]+1;
    val += corner_value[0];
    put_field (*f, HOME, (voidptr) &val);
  } else if (coords[Y]==max[Y]) {      /* right */
    val = (corner_value[2] - corner_value[1]);
    val *= coords[X]-min[X];
    val /= max[X]-min[X]+1;
    val += corner_value[1];
    put_field (*f, HOME, (voidptr) &val);
  }
}
```

```
void set_boundary_sites(set s, float *corner_value, field f)
{
  do_task(set_boundary_sites_, s,
  PASS, corner_value, FOUR*sizeof(int),
  PASS, &f, sizeof(field),
  END);
} /* set_boundary_sites */
```

The final task routine shows how the same return value can lead to radically different answers depending on the keyword in the do_task triplet. get_limits returns the value of field f on the HOME site twice, but the do_task integrator only keeps the maximum value of the first argument and the minimum value of the second. The purpose of this routine is to return the maximum and minimum values of the field on some set of sites.

```
void get_limits_(float *high, float *low, field *f)
{
   *high = *(float*)field_pointer (*f, HOME);
   *low = *high;
}
void get_limits(set set_of_sites,
                float *high,
                float *low,
                field f)
      /* high and low must both be pass-by-address, */
      /* since they are return values               */
{
   do_task (get_limits_, set_of_sites,
            MAX_REAL, high, sizeof(float),
            MIN_REAL, low, sizeof(float),
            PASS, &f, sizeof(field),
            END);
}
```

3.5 Control Routine

The main entry point of a Canopy program is a subroutine named `control()`. This is executed as an ordinary C program on a single node of the system. Only when `do_task()` is called does Canopy start parallel processes. There are therefore some distinctions between the `control` program level and the task routine level. Most importantly, task routines cannot call other task routines (though there is a sub-task facility using `maps`, it does not re-distribute control to another node) and cannot do I/O. The `control` program may call other subroutines that are also part of the control section—task routines may only be entered through `do_task` and `do_task_n_times`.

Technically speaking the task routine headers are part of the control section since they do run at the top level and call `do_task`, but it is much more convenient to to keep them near the task routine itself as has been done in `laplace.c`.

The only other subroutine in the control section of the Laplace program is `show_field`, which prints out the field on a grid using the digits `0` through `9` to indicate the value of the field normalized to its limits. It uses the `get_limits` task to find the upper and lower limits of the field value and loops over all the sites on the lattice the hard way to print them:

```
void show_field(grid g, field f)
{
    int *max = grid_upper_bounds(g);
    int *min = grid_lower_bounds(g);
    site s;
    coordinates coords;
    int i;
    int j;
    int k;
    float a, h, l; /* h and l are upper and lower limits */

    get_limits (g, &h, &l, f);

    /* loop over all sites the primitive way  */
    /* one site at a time on the control node */
```

```
    for (i=0; i<(max[X]-min[X]+1); i++) {
      for (j=0; j<(max[Y]-min[Y]+1); j++) {
        coords[X] = i;
        coords[Y] = j;
        s = site_at_coordinates(g, coords);
a = *(float*)field_pointer (f, &s);
k = 9.99/(h-1) * (a-1);
printf ("%11d",k);
      }
      printf("\n");
    }
} /* show_field */
```

The control program itself is the main entry point for a Canopy program. It is called by the system after all of the Canopy set-up completes. Notice particularly how the size of the grid plane is input before being used to declare the grid—since `complete_definitions()` is an executable routine, Canopy provides flexibility in changing sizes at run-time.

```
void control ()
{
  grid plane;     /* declare the grid, field and  */
  field points;   /* set variables -- note they   */
  set rb_sites;   /* are local.                   */
  set boundary_sites;
  set red_interior;
  set black_interior;

  int x_dimension;      /* these are the values   */
  int y_dimension;      /* input by the user.     */
  int number_of_sweeps;
  float corner_value[4];
  float a0,a1,a2,a3;

  int i, j; /* every program needs a few integers */

  /* Now input the size, boundary values, and     */
  /* sweeps using routines in the prompt library  */
  /*                                              */
  prompt_scanf(2,"Enter dimensions height, width: ",
```

```
            "%ld,%ld",&x_dimension,&y_dimension);
number_of_sweeps =
   prompt_int("Enter number of sweeps: ");
prompt_scanf(4,
      "Enter 4 corners clockwise from top left: ",
      "%f,%f,%f,%f", &a0, &a1, &a2, &a3);
printf("Enter 1 to read file or 0 for new run: ");
i = prompt_int
      ("Enter 1 to read file or 0 for new run: ");

corner_value[0] = a0;/* a0 ... a3 are just con- */
corner_value[1] = a1;/* venient abbreviations.  */
corner_value[2] = a2;
corner_value[3] = a3;

/* Now call the declaration routines.  Notice   */
/* how there were C executable statements before*/
/* these calls to get input values.             */
/*                                              */
plane = periodic_square_grid (x_dimension,
                              y_dimension);
points = site_field(plane, sizeof(float));
rb_sites = set_of_sites(plane, rb_interior);
red_interior = set_of_sites(plane,
                            red_interior_func);
black_interior = set_of_sites(plane,
                            black_interior_func);
boundary_sites = set_of_sites(plane,
                            boundary_func);

/* Now set up the grids, fields and sets with   */
/* this call to complete_definitions.           */
/*                                              */
complete_definitions();

/*                                              */
/* The first tasks are to zero the field points */
/* ary values.  Each of these tasks is done in  */
/* parallel on all the nodes.                   */
/*                                              */
set_field_to_zero(plane, points);
set_boundary_sites(boundary_sites,
```

```
                    corner_value,
                    points);

  /* Now open the field file if the field was to  */
  /* be read.                                     */
  /*                                              */
  if (i==1) {
    open_field_file("lap.out",READ);
    read_field("lap.out",points);
    close_field_file("lap.out");
  }

  /* Now print the initial values -- note that    */
  /* show_field must loop over the sites itself.  */
  /*                                              */
  printf("Initial values: \n\n");
  show_field(plane, points);

  /* Now sweep over the grid plane                */
  /* number_of_sweeps times.                      */
  /*                                              */
  /* IMPORTANT:                                   */
  /*   It would appear on the surface not to be   */
  /*   necessary to red-black the laplace sweep.  */
  /*   Unfortunately, in a real multi-processor   */
  /*   system it is not guaranteed that one might */
  /*   hit the timing exactly wrong and try to    */
  /*   read a field while another node is updating*/
  /*   it. For fields of length greater than one, */
  /*   this may result in reading a half-updated  */
  /*   field. This can lead to improper results   */
  /*   even if either the old or the new field    */
  /*   were by itself permissible.                */
  /*                                              */
  for (i=0; i<number_of_sweeps; i++ ) {
    laplace_sweep(red_interior, points);
    laplace_sweep(black_interior, points);
  }

  /* Now print out the final values               */
  /*                                              */
  printf("\n\nFinal Values:\n\n");
```

```
  show_field(plane, points);

  /* and write them to a field file.              */
  /*                                              */
  open_field_file("lap.out",WRITE);
  write_field("lap.out",points);
  close_field_file("lap.out");

} /* control */
```

3.6 Running the Example Program

A Canopy application can be run on a massively parallel system such as ACPMAPS, or on a single-node Unix machine. Individual Canopy User's Guides are provided for massively parallel systems. Here, we illustrate how the `laplace.c` application would be run on a Unix system.

The Canopy-specific tool `canc` is used instead of `cc` to compile and link the Canopy program:

```
>canc laplace.c -o laplace
```

The `laplace` module is an ordinary UNIX modules which runs in the ordinary UNIX way:

```
>laplace
Enter dimensions height,width: 6,58
Enter number of sweeps: 1000
Enter 4 corner values clockwise from top left: 1,2,3,4
Enter 1 to read file or 0 for new run: 0
Initial values:
22222222222333333333333333333333333444444444444444444444
30000000000000000000000000000000000000000000000000000005
40000000000000000000000000000000000000000000000000000005
60000000000000000000000000000000000000000000000000000006
70000000000000000000000000000000000000000000000000000006
99999999999999999999988888888888888888888877777777777777
```

```
Final Values:

Largest site value is 4.000000 at (5,0)
00000000000000000011111111111111111122222222222222233333
11122222222222222222222222222333333333333333333333333333
33333444444444444444444444444444444444444444444444444444
45555555555555555555555555555555555555555555555555555554
67777777777777777777777777776666666666666666666666666655
99999999999999999888888888888888877777777777777777666666
```

Chapter 4

Parallelism Considerations

Canopy is designed to work in a parallel computing environment consisting of many independent processors with independent memory address spaces. The `control` program is run on one of these processors and the task routines are distributed across all of them. Most of the work involved in doing this is automatic but there are a few special considerations of which users should be aware.

In general, when a task is done over a set of sites, each site can be viewed as a virtual processor: The task routine is the program, and the `HOME` site's field data (along with any stack space used) is the local memory. The paradigm is that all sites are done simultaneously; a practical consequence is that any operation that depends on the order of processing sites within a simple task is logically unsound. (For compound tasks, operations which depend on the order of sites within a given level would be unsound.)

4.1 How Field Pointer Works

The `field_pointer` routines look innocent enough, but on close examination it is obvious something is missing. On a single CPU with a single memory space, the address of any object can always be returned, but on a distributed system the field element may be in some other

node's address space. If the field element is off-node, a *copy* of that element must be made into local memory; `field_pointer` then returns the address of the *copy* to the caller. This has several implications:

> The returned pointer may point to either the original field element or a local-memory copy of the field element
>
> Something allocates memory for the copy
>
> Either something frees memory for the copy with no user intervention or wasted memory piles up somewhere

The first implication means that the field element must be considered read-only *unless* it is certain that it was on the same node as the call. As the user has no way of telling which node is which, *all* of the elements must be considered read-only. There is one exception to this designed to improve efficiency in some common cases: inside task routines the `HOME` site is certain to be on the current node. This means that fields on the `HOME` site may be updated without `put_field()`. IMPORTANT: This is only true in task routines called by `do_task()` and `do_task_n_times()`. In *sub-task* routines called by `do_task_on_inverse_image()` the `HOME` site is not guaranteed to be on the current node.

The memory management issue is more interesting. Since pointers inside *task* routines only are valid inside the task routine (and may not be returned since it makes no sense to combine the values returned by each site into a single value), the memory used for off-node field storage can be returned to the pool after each call `do_task` makes to the task routine. There is a special area of memory for this (on each node) called the *do_task lalloc heap* which is set up at the beginning of the program. However, too many calls to off-node `field_pointer` reads inside a task may still overflow the lalloc heap. There are two ways to deal with that situation. One is to call the `declare_lalloc_sizes()` routine in the declaration section of the `control` program to make the do_task lalloc heap larger, which is generally best unless the number of calls can become large and variable. The other is to call `reset_lalloc()` from inside the task routine itself, invalidating *all* pointers previously

obtained through `field_pointer` by freeing the *entire* lalloc heap at once. Any field element that needs to survive a call to `reset_lalloc()` must be copied to a new area before the call. At the present time there is no selective `free_lalloc` routine which determines whether a pointer points to the lalloc heap and then frees just that area if it does. In the future such a routine may be implemented, but for now `reset_lalloc` is the only way to free space.

In the `control` program things are worse since pointers in the `control` program and its subroutines are theoretically valid forever. This could require an infinite amount of storage on the control program. The solution Canopy adopts is to have `field_pointer` class routines from the `control` program use memory from another area called the *control lalloc heap.* Unlike the do_task lalloc heap, pointers allocated on the control lalloc heap are valid for all time. Since many programs call `field_pointer` routines once or more per *iteration,* it is often impossible to fix overflows by extending the heap with `declare_lalloc_sizes()` . Instead, the `reset_lalloc` routine must be used with its attendant headache of invalidating *all* old data. The best way to do this is to call resetlallocx at the end of each iteration if the program calls `field_pointer` routines from the `control` level. Note that calls to `reset_lalloc` inside a task routine do not affect the control lalloc heap.

A side issue concerning the lalloc heap is that of data alignment. For some CPU chips, the i860 in particular, it is crucial that double precision numbers be 8-byte aligned. Canopy handles that properly by having lalloc return 8-byte aligned addresses whenever a multiple of 8 bytes is requested. (The malloc routine in C works the same way.) Since fields containing double precision data should always be sized in multiples of 8 bytes (assuming that a sizeof() is used as `nbytes` in the field definition routine), the alignment of data fetched by `field_pointer` should always be proper.

4.2 (Lack of) Global Variables

The `laplace.c` program does not have a single global variable. This is not due to some perverse stylistic purity concept of its author, but rather to a sad fact of life on parallel machines: what C thinks is a global variable is not really a global variable. This confusion arises because each of the nodes has a separate address space, and if C declares a global variable an *independent* copy is allocated on each node. Updating a global variable on one node will not update it on another, and if two nodes try to update the same physical memory location by using full-addresses at the same time bizarre synchonization errors may occur. As a result, global variables can lead to many strange bugs and should be avoided wherever possible.

There are a few cases where global objects can be used safely. First, the field elements and Canopy data are global variables of a sort, but they are always used through subroutines with special rules which keep the data sensible. Second, global variables can be set in the `control` program, and the values broadcast to all the nodes. Such variables are read-only from the perspective of task routines. Third, global variables can be used to transfer data between subroutines running on the *same* node, such as to pass values between various subroutines in the `control` program.

Global variables might also be used to pass values between subroutines called by a particular task routine. This use of "task global" variables is not encouraged. Since the space of globals is shared by all the sites processed on a node, using task global variables contradicts the paradigm of each site being an independent virtual processor. Consequently, these variables must be handled in a special way if "multi-thread" is used to improve communications efficiency—see the section on TASK GLOBALS. Similar issues arise on systems where multiple processors share the same address space—global variables which are "one copy per processor" must be distinguished from those which are really global.

There are some common temptations to use global variables where they won't work at all. Canopy provides structured ways to handle

these situations. For example, to accumulate values inside tasks use integrate arguments instead of updating a global variable. The global variable method will not work when running on more than one node. Another improper use of global variables is to synchronize tasks by setting a global variable flag. Instead, the compound set of sites and synchronization routines will do the job correctly.

A valid use for global variables is to set up tables of initial values or pass overall parameters. The `broadcast()` subroutine can set this up *providing* that the variables are treated as read-only after being `broadcast`. A call to `broadcast` makes a static global variable have the *same* value on *all* nodes that it does on the control node. For example:

```
int var[4];
...

void control() {
...

var[0] = 2; var[1] = 5; var[2] = 11; var[3] = 17;

broadcast( (voidptr) var, sizeof(var) );
...

} /* control */
```

After the `broadcast` call the values 2, 5, 11, and 17 are in `var` on every node in the system, so task routines can read `var` as if it were an ordinary global variable. Notice however that no routine can *change* `var` after the call to `broadcast` without causing total confusion. It is important to remember that `broadcast` only works on global static variables, because only for global static variables is the same memory address allocated on each node. If a call to `broadcast` is made with an automatic variable or one for which space has been dynamically allocated (say with `malloc`), the results will be unpredictable.

4.3 Arguments to Task Routines

There are logical subtleties associated with passing arguments to task routines. The first problem is that of length of data. In C, it makes sense to pass an argument by address; it is assumed that the function will use the data in the appropriate manner. But Canopy platforms are allowed to have distributed memory, so a copy of the data representing the argument must be sent to each processor node running the application. This means that somehow the Canopy routines must be made aware of the length of data that the task routine will be using.

The use of arguments to return values from a task routine presents another issue. In C, the calling routine can receive a return value by passing the address of a variable—the subroutine will eventually store a result at that address. But what does it mean to have each of many sites return a value? If an array of answers is required, Canopy already provides the concept of "one datum for each site": A `field`. If a single value is expected, there is the question of how to amalgamate the values returned by the task routine for each site, into that single value. The routine may wish to return the sum of the values, or the maximum value, or some other function of the values. Therefore, for each argument to a task routine invoked via `do_task`, one must specify the data length and how to handle the argument, along with the argument's address. The syntax Canopy uses to do this is that of `do_task` triplets, described in Section 6.2.1 DO_TASK ROUTINES.

Canopy does not support two-way arguments, where the user passes a value to the function and then expects a return value at the same address. Also, there is no return value for the `do_task` routine itself.

Having resolved the above issues, the use of task routine arguments is not inefficient—the same procedure that handles the start and finish of task routines can handle the arguments. Except for overall (one-time-only) initialization, the use of arguments to task routines is recommended over the naive alternative of using (and explicitly broadcasting) global variables.

4.4 Synchronization

The most obvious way the parallel nature of the computer intrudes into Canopy programs is by requiring that calls to task routines not depend on the order sites are processed. Even if the algorithm is insensitive to order it may still be inappropriate to let `do_task` update all the sites on the grid at once. For example, `laplace.c` must use an explicit red/black order for the reason discussed below.

Consider what happens when a Canopy program runs on more than one processor. The `control` sections are executed by one processor (called the *control node*) and then the task routines are done in parallel on all the processors. If the Laplace update algorithm were run with the set of all interior sites the following error could occur: one processor could have written half of the field value it updates at the same time another processor reads that in to update a neighboring site. This causes the update on the second processor to be wrong *even though the* order *of the update is immaterial.* Fortunately this and other synchronization errors may be avoided by following these two rules:

> Rule 1: *Always write tasks so they only update field elements on their* `HOME` *site.*
>
> Rule 2: *Arrange sets and* `do_task` *calls so that if a task updates a field element on its* `HOME` *site it never reads elements of that field from any other site in the set currently being processed.*

For the Laplace example this means that `update_site` cannot read the field elements of `*f` from any site in the set being processed by the `do_task` call, but may read elements of `*f` from sites not in that set. Because all the neighbors of red sites are black sites and *vice versa*, doing the job in a red-then-black order solves this problem.

Of course, doing separate `do_task` calls for each set has some disadvantages: notably the number of sets may increase dramatically and the opportunity to update sites out of strict order by waiting only for their neighbors to finish, instead of the entire set, is lost. The *compound* set

of sites, in conjuction with the sync_field_pointer routines, provides a way to take advantage of parallelism without such strict order rules but adds some complications of its own. For example, the laplace.c program could use a compound set of sites to do the red-then-black updating all at once. Such a compound set, called rb_sites, was defined. However, more must be done than just replace the two do_task calls with red_interior and black_interior with one using rb_sites. The update_sites routine must also be modified as follows to use sync_field_pointer():

```
void laplace_sweep_(f)
field *f;       /* always pass-by-address in tasks */
{
  float a =
  *(float*)sync_field_pointer_at_dir(*f, X) +
  *(float*)sync_field_pointer_at_dir(*f, Y) +
  *(float*)sync_field_pointer_at_dir(*f, MINUS_X) +
  *(float*)sync_field_pointer_at_dir(*f, MINUS_Y);
  a /= 4.0;
  put_field (*f, HOME, (voidptr) &a);
}
void laplace_sweep(set_of_sites,f)
field f;      /* note that f is pass-by-value here */
set set_of_sites;
{
  do_task(laplace_sweep_, set_of_sites,
          PASS, &f, sizeof(field),
          END);
} /* laplace_sweep */
```

The calls to sync_field_pointer will wait for the task routine at the target site to finish if the target site is at a lower level then the HOME site. Here is a rule to decide where to use the sync-class routines:

> Rule 3: *Use the* sync *routine only to fetch field elements of fields whose elements on the* HOME *site are* updated *by the task. Fetch all other field elements without the* sync *routine.*

Note that there are certain situations, such as where a field update requires some new elements and some old elements, where this system is inadequate. These should be handled by multiple sets of sites. When a task is done over a compound set of sites the calls to the task routine will not necessarily be done in order, and it is not even guaranteed that all sites on a particular level will finish before sites on other levels start (in fact, that is part of the point!). Since it is easy to make mistakes here it is strongly recommended that programs be written without using compound sets and sync routines and then modified after they are working. That way any errors due to the modification will show up clearly.

4.5 Note on Efficiency

The `laplace.c` program illustrated in the TUTORIAL EXAMPLE is, as it stands, not very efficient (if efficiency is measured by megaflop rate) because it spends most of its time fetching numbers instead of doing arithmetic. On realistic lattice gauge problems this sort of arrangement works better because proportionally more computation is done on each field. For example, if the Laplace program were re-written so that each field element had a hundred numbers instead of one then the time spend fetching fields would be less significant.

The `field_pointer_from_address()` and `address_of_field()` routines are available to create and use pre-computed field addresses and thus avoid the overhead of computing locations on every sweep, but this optimization does not save much time. In absolutely critical applications they may make a 5% or so difference (and they aren't always faster).

In general, the efficiency of a Canopy application on a given system will depend on several factors, including: the fraction of work which is done by tasks (as opposed to the control program); the expense of the Canopy "bookkeeping" to fetch fields and move among sites; the cost of synchronization in terms of nodes becoming idle during and at the end of tasks; and the cost of internode communication required to

transfer data between processors.

For realistic problems running on typical massively parallel systems, most work is done in tasks, and the bookkeeping and synchronization inefficiencies are small. Canopy bookkeeping becomes expensive when very little work is done by each task. Synchronization inefficiency can occur if each node is handling only a few sites, or on algorithms with compound tasks implementing an order of processing which forces sequential operation.

4.6 Multi-thread

A number of strategies have been implemented in the Canopy library to improve performance of communications-intensive Canopy applications; these strategies are collectively referred to as "multi-thread", since they involve the running of multiple threads of execution on each node.

Multi-thread operation improves performance because it allows many small transfers to be transparently coalesced into fewer, larger transfers. Fewer communications overheads are taken, and the available communications bandwidth is used more efficiently. This coalescing is done by processing multiple sites in parallel on each node. Each site has its own CPU context—multi-thread is simple form of multi-tasking. The coalescing of data transfers is done transparently by the Canopy library, by grouping together transfer requests from multiple sites.

In most cases, no modifications are required to existing Canopy applications, to allow them to be run multi-threaded. The Canopy library guarantees bit-for-bit identical results for correctly written applications, whether run single- or multi-threaded.

The use of multi-thread can be controlled by Canopy subroutine calls and command-line switches to the canopy hosting tool. These switches are explained in the RUNNING APPLICATIONS section of the CANOPY ACPMAPS USER'S GUIDE.

4.6.1 How Multi-thread Works

In "classic" Canopy, task routines are single-threaded. A task routine which requests off-node data (via a `field_pointer` request) blocks—no other work is done on the node requesting the transfer until the data transfer completes. Likewise for `put_field()`—all processing on the node is suspended until the data is actually written to the remote node's memory.

Under multi-thread operation, multiple sites are processed in parallel on each node; each site being processed has its own CPU context (thread). When `field_pointer` requires an off-node field access, the CANOPY run-time library records information about the requested transfer, but *does not actually perform the remote access.* Instead, the CPU moves on to do the task routine for another site—another "thread" of execution. The new thread has its own stack, but is running the identical task routine; the cost of this "context switch" is minimal.

This process is repeated until all of the available threads are waiting for off-node data (or the list of sites to be processed is exhausted). Then, all of the requests for each target node are combined into a single scatter/gather transfer request. The transfers are done, and threads which now have the data they need are allowed to proceed. The key point is that of the many access requests accumulated, usually several will involve the same remote node — these requests can be coalesced into a single transfer.

Similarly, off-node writes required by `put_field` are queued up; in this case the same thread continues execution, until a remote read access or the end of the task forces it to do the remote writes.

Multi-thread gains when the cost of a thread context switch is smaller than the overhead associated with a transfer, or when communications overheads affect the availability of a resource which is shared among many nodes. So any advantages depend on the nature of the system, and on particulars of the algorithm and its communications pattern. On the ACPMAPS system at Fermilab, when production lattice gauge physics programs are run on hundreds of nodes, the typical

application gains a factor of 2–3 in performance with multi-thread enabled.

4.6.2 Control of Multi-thread

To use multi-thread mode, all that is generally necessary is to specify the number of threads to be used; this controls how many sites can be processed in parallel. The number of threads may be set between 1 and MAXTHREADS, which is currently 512 on ACPMAPS. Generally, the more threads the better for efficiency, but each thread requires some extra memory. Each thread requires a separate stack; for typical Canopy applications, the default value of 8K bytes of stack per thread is adequate. Each thread also gets its own local allocation `lalloc` heap (see the section 6.3.9 THE LALLOC HEAPS in THE CANOPY SUBROUTINE REFERENCE); if a non-default size for the task lalloc heap is declared, this is applied to every thread. (The default `lalloc` heap size is 2K bytes.)

By default, the number of threads is set to 1 and Canopy processes each site on each node sequentially. The number of threads may be selected at the start of the job by the `-threads <nthreads>` option on the canopy command line or via the environment variable `CAN_THREADS = <nthreads, nstack>`. It may also be changed from within the application by calling the function

```
multithread(int nthreads, int stack_size);
```

from the control program. This may be called multiple times in an application to dynamically change the number of threads or stack size, but it is a relatively expensive operation and so should not be done frequently. It is best to choose a number of threads and stack size which works for the entire application, and set them once at the start of the program.

For optimal performance, the user must choose an appropriate number of threads. The additional memory used for stacks and local allocation heaps is about 10K bytes per thread, so if the application is tight on memory, that can restrict the number of threads. In general,

the more threads the better; but adequate transfer coalescing is usually achieved by the time `nthreads` reaches 32–64.

Under certain special conditions it may be desirable to temporarily disable context switching between threads. This might be useful if:

> A section of code which does its own synchronization needs to protect a critical section (i.e. to prevent other threads from running, even if the current thread requests an off-node access).
>
> It is "known" that a task on some site is critical—many other sites are waiting on results from the current site. In this case allowing the task on the current site to finish quickly, at the expense of other sites on the same node, can be a net win.
>
> A task violates the multi-thread rules for usage of task global variables or does something else unorthodox.

Two functions, `multithread_disable()` and `multithread_enable()`, are provided to disable and re-enable multi-thread context-switching.

4.6.3 Advanced Multi-thread Features

This section describes techniques which may be useful in certain special situations — dealing with non-standard usage of global variables, fine-tuning applications for maximum efficiency, and getting statistics on multi-thread coalescing and memory usage.

Task Globals

It is possible (but considered harmful) to use global variables to pass values between subroutines called by a particular task routine. For example:

```
int tg;          /* A task global    */
mytask_ ()       /* The task routine */
{
  SetThingsUp ();
  ...
  mysub ();
  ...
}
SetThingsUp ()
{
  ...
  tg = whatever;/* Set global to its desired value*/
}
mysub () /* Subroutine done during this task */
{
  value = tg;   /* Using the value set up */
}
```

Here, the variable tg is written in one subroutine and used in another; we call tg a "task global". The light-weight context-switching used by multi-thread does not provide a separate copy of task globals for each thread. This means that the task globals, unlike broadcast globals and automatic (stack) variables, cannot be used if nthreads > 1.

The following work-around may be used in cases where the task global construct is required: during execution of a task routine with multi-thread enabled, the CANOPY run-time library guarantees that the global variable CAN_my_thread will contain a unique value, between 0 and MAXTHREADS-1. This value will be unique in each active thread. An appropriately defined array can replace a task global, as illustrated in this example:

```
int tg_[MAXTHREADS];
#define tg (tg_[CAN_my_thread])
/* Now there is a separate tg for each thread */
mytask_ () {    /* The task routine */
{
  SetThingsUp ();
  ...
  mysub ();
  ...
}
SetThingsUp ()
{
  ...
  tg = whatever;
  /* Expands to tg_[CAN_my_thread] = whatever */
}
mysub () /* Subroutine done during this task  */
{
  value = tg;
  /* Expands to value = tg_[CAN_my_thread]    */
}
```

A warning: The above technique has several limitations. The use of `#define` to hide the indexing operation becomes more complicated if the task global is an array or a structure, and will fail if the name of the task global matches the name of an element in some other structure. Also, task globals take up at least `MAXTHREADS` times as much space as before, and become slightly less efficient to access. (NOTE: For cases where the actual number of threads used tends to be much less than `MAXTHREADS`, some savings in memory usage may be realized by dynamically allocating the array with the `malloc()` library function, such that it has exactly `CAN_nthreads` elements, instead of `MAXTHREADS` elements.)

Independent of Canopy considerations, using global variables in this way may be considered "poor coding practice"; but if there are compelling reasons, they can be handled (with care) as illustrated above.

Vertical Coalescing

Normally, the thread associated with a given site will block when it calls `field_pointer()` to request off-node data; the thread resumes execution only after the data has been obtained. There is a small amount of overhead incurred whenever this happens, due to the thread context switch. In some situations, it may be possible to reduce the number of context switches (thereby increasing program efficiency), by combining several `field_pointer()` calls made from the same thread. This is called vertical coalescing.

The CANOPY library provides routines which allow vertical coalescing to be explicitly turned on and off. All `field_pointer()` calls sandwiched between `multithread_begin_vertical()` and `multithread_end_vertical()` will be coalesced. IMPORTANT: This means the returned pointers will not be valid until after the `multithread_end_vertical()` call. Here is an example:

```
...
multithread_begin_vertical();
/* queue the requests but do not do the actual transfers */
for (i=0; i<n; i++) {
  ar[i]=(whatever*)field_pointer(f1, &sites[i]);
  /* *(ar[0..n-1]) is not valid here yet */
  br[i]=(whatever*)field_pointer(f2, &sites[i]);
}
multithread_end_vertical();
/* now ar and br are valid */
...
```

Because off-node accesses after a call to `multithread_begin_vertical()` might not receive data until `multithread_end_vertical()` is executed, routines which return a `site` structure which might be based on off-node information are illegal while vertical coalescing is active. Specifically, `move_site()`, `move_site_by_path()`, and `site_at_path()`, are illegal between calls to `multithread_begin_vertical()` and `multithread_end_vertical()`.

It is a logical error for a task to terminate while vertical coalescing still active, since any field pointers that have been acquired have never

been valid. Canopy will declare an error and terminate the job if this happens.

NOTE: When multi-thread is enabled, vertical coalescing is *always* enabled for data written to remote nodes with the `put_field()` function. Vertical coalescing of `put_field()` calls is not affected by `multithread_begin_vertical()` or `multithread_end_vertical()`.

Copying Put_field() Data

Most Canopy programs use `put_field()` only to modify field data at the HOME site, but some (for example, the FFT routines in FFTLIB), modify data at other sites. When multi-thread is enabled, outgoing data transfers are coalesced: `put_field()` requests are queued, and the thread is allowed to continue. All `put_field` requests are buffered until processing of the site is complete; at that time, all of the buffered `put_field()` requests are flushed.

It is possible for the current thread to modify the contents of the buffer passed to `put_field()`, before completing the task at the current site. If this happens, there is a danger that the wrong data will get written to the remote node, since the data has been changed before the queue of outstanding `put_field()` requests has been flushed. To avoid incorrect behavior in this situation, the `put_field()` function normally makes a copy (on the `lalloc` heap) of any data being sent off-node. Making these copies imposes a small performance penalty, and increases the required lalloc heap size. (Since each thread has its own lalloc heap, the amount of additional memory required can be fairly large.)

Two routines are provided, which allow a task routine to control whether or not a copy of off-node `put_field()` data is made. The `multithread_begin_nocopy()` function tells Canopy that it is safe to skip the copying of `put_field` data, since the contents of these variables will not be modified until after a call to `multithread_end_nocopy()`. When `multithread_end_nocopy()` is called, the queued `put_field()` requests are flushed. These routines will normally be used to bracket a short section of code which issues a series of `put_`-

`field()` requests, or a loop, which writes out the results of some operation.

Miscellaneous Multi-thread Interactions

The features available for control of multi-thread opoerations are intended to provide performance advantages when used in the simple intended ways. For instance, veritcal coalescing should be invoked by sandwiching a short section of code (containing `field_pointer`-class routines) between calls to `multithread_begin_vertical()` and `multithread_end_vertical()`. Similarly, `multithread_begin_nocopy()` should start a short code segment which is terminated by `multithread_end_nocopy()`. Invoking multi-thread should never alter the result of a program.

However, Canopy cannot prevent these routines from being called in arbitrary combinations and circumstances. In these cases, the prime object is that the Canopy program will always get the correct result. In order to prevent errors, certain combinations of advanced options will be "turned off", behaving in a conservative manner with respect to waiting and coalescing transfers:

> Calling `field_pointer()` automatically causes the queue of pending `put_field()` requests for the current site to be executed. This is necessary to prevent data coherency problems which could otherwise occur if a task writes data with `put_field()`, then turns around and attempts to access the same data again using `field_pointer()`.
>
> A task or subtask inherits the multi-thread state (enabled or disabled) of its parent. Furthermore, if multi-thread is disabled on entry to a task or subtask, then that task or subtask may not itself enable multi-thread (any calls to `multithread_enable()` are ignored).
>
> All tasks and subtasks started while multi-thread is enabled begin executing with vertical coalescing disabled, and copying of `put_field()` data enabled. Calling a subtask does

not affect the state of these settings in the calling task, but will have the side effect of causing the queue of pending off-node requests (both read and write) for the current site to be executed.

Calling any of the following routines, or returning from a task routine, will cause any queued `put_field` or `field_pointer` requests for the current site to be executed:

```
multithread_enable()
multithread_disable()
multithread_begin_vertical()
multithread_end_vertical()
multithread_begin_nocopy()
multithread_end_nocopy()
```

Multi-Thread Statistics

The `print_multithread_stats()` function prints statistics which may be useful for fine-tuning the number of threads and/or stack size for an application. It reports the average degree of communications coalescing, and stack usage statistics. The statistics are reset each time `print_multithread_stats` is called. The `print_multithread_stats` function may be called only from the control program (it is illegal inside of tasks).

4.7 General Multi-Processing Issues

4.7.1 Random Numbers

There are several subtleties relating to the use of (pseudo-)random numbers in a massively parallel environment. For one thing, the increased computational power of these systems means that many more random numbers can be run through; this implies more severe requirements on the quality of the random sequence. Another point is that it is desirable to have precisely reproducible results; many schemes for generating random numbers would lead to different results depending on the number of nodes used. And it can be undesirable for two streams of random numbers used in the same job to actually be two parts of the same longer stream—this increases the likelihood of getting invalid results due to similarities between the streams.

Even in single-processor systems, there are examples of researchers obtaining incorrect results by using flawed random number generators. Although Canopy provides a generator which is strongly felt to have excellent properties (see RANLIB—RANDOM NUMBERS), the user is free to provide a different kernel for generating the random sequence, within the bookkeeping framework Canopy provides.

The proper logical entity for producing a stream of random numbers is the site (the virtual processor). If a separate stream is associated with each site, then random numbers will not cause results to depend on the number of nodes, or on how the sites are distributed among the nodes. Since typical jobs involve millions of sites, there are concerns about correlations among the millions of streams, as well as the usual concerns about randomness of an individual stream.

To set up and use random numbers in Canopy, the progam calls `make_random_generator`during the declaration phase—the random function is specified here. (Canopy provides in RANLIB an excellent function: `dual_random`, which is based on a large feedback register method. However, the user is free to create a different function—see section 6.1.5 RANDOM NUMBER DECLARATION.) Whenever a random number is desired, whether during a task or otherwise, the `random()` function is

called.

The naive way of handling randoms numbers on massively parallel systems is to have one stream of numbers per node. Compared to stream-per-site, this approach is very slightly faster, and saves the necessity of allocating room for seeds and queues of randoms for every site (about 80 bytes per site). Canopy provides the option of stream-per-node randoms. Using stream-per-node has the consequence that repeated runs of the identical application on different numbers of nodes will *not* produce identical results. If multi-thread is active, stream-per-node randoms can lead to non-identical results, even if the number of nodes is unchanged. Identical repeatability is often useful, so most applications should select stream-per-site random numbers.

4.7.2 System Independence

In important consideration in program development is that programs should give bit-for-bit identical results regardless of details such as the number of nodes or the exact CPU on which the program was run. Unfortunately, while Canopy was designed with this principle in mind, C was not. However, if stream-per-site random numbers are used, the only remaining problems are related to floating-point arithmetic. The following may cause programs to give results which are not bit-for-bit identical:

> *Integrate arguments involving floating point operations are used in* `do_task`, *and the number of nodes has changed.*
>
> *The compiler has re-arranged the order of floating-point operations.*
>
> *The supplied C math library, or the floating point arithmetic itself, is different. (Not every CPU is perfectly IEEE compliant.)*

Only the first of these is specific to Canopy. If stream-per-node random numbers are used, the the following discrepanceies can also occur:

> *Random numbers are used, and the number of nodes has changed. Since different streams of randoms will apply to a given site, the results can be completely different. The difference is analogous to starting with a different random seed.*
>
> *Multi-thread context-switching is active, and randoms are called for during task routines after* `field_pointer` *is called. In this case, the order of randoms can vary even if the number of nodes remains unchanged.*

Of course, a genuine error can cause inconsistency (and invalid results). This can happen if the application violates the Canopy paradigm, which states that sites in a task are (logically) processed simultaneously. For example, if the `laplace.c` example did not use the red-black technique to avoid the use of "stale" field data, it could produce inconsistent results, depending on just how quickly each node does its work. Or the application might use a global variable which is set but not broadcast, or broadcast a variable which is obtained through dynamic allocation (and in different places on various nodes).

Chapter 5

Typedefs, Structures, and Variables

This chapter and the next chapter CANOPY SUBROUTINE REFERENCE, form a reference manual for the Canopy software.

Canopy is designed as a set of subroutines callable from ordinary C programs. This chapter describes the typedefs, structures, #defines and global variables visible to the user and explains how to use them. These objects are all declared in the include files for the Canopy system. There are a few additional reserved words in Canopy which are private variables and functions which for some reason or other cannot be hidden. These reserved names all start with the string "CAN_" so they can be avoided by normal programs. There are additional private structures and typedefs but these are not in the linker tables so they do not affect user programs.

Canopy is a layered product, with each layer presenting a clean interface to the higher layers. The lowest layer, the Canopy Hardware Interface Package (or CHIP), is designed to isolate the higher layers from the machine and system details. The next layer, Canopy, is built upon CHIP; other concept-oriented tools can be built upon the same CHIP foundation.

Canopy is a set of subroutines designed for solving grid-oriented problems on a multi-processor machine. Accordingly, the canopy.h

include file contains structures and typedefs for grid, field, and site variables; and function prototypes for the routines that manipulate them. Everything at the Canopy layer is built upon CHIP, so the site variables, for instance, know about multiple nodes. However, a great deal of effort has been spent ensuring that the user does not need to know about CHIP; the multi-processor (or not) nature of the real machine is not visible from the application software.

All the CHIP features documented here are visible at all levels. The include file `canopy.h` itself includes `chip.h`, so the casual user need never be concerned about the levels at which features are defined: As long as `canopy.h` is included and the application is compiled using a Canopy compilation tool (to link in appropriate libraries), an application can freely mix Canopy routines and direct CHIP primitives if needed. Normally, CHIP routines will not be called directly; but some typedefs defined at the CHIP level (e.g. `voidptr`) will commonly appear in user applications.

For convenience in porting Canopy to other platforms, and in identifying the nature of the CHIP interface, concepts defined at the CHIP level are identified explicitly in this chapter. The last section of the CANOPY SUBROUTINE REFERENCE forms a reference for CHIP subroutines.

5.1 Typedefs

CHIP sets up some simple types to allow the ANSI-style Canopy code to work on non-ANSI compilers. It is recommended that user Canopy programs utilize these same types (e.g. `voidptr` rather than `void*`) so that the applications will be portable to Canopy platforms which do not have ANSI C.

`void`: `void` is defined as `int` (with a macro definition) if a non-ANSI compiler lacks it, so programs can, with no modification, take advantage of ANSI type-checking for void function return values when compiled with an ANSI compiler.

`logical`: A synonym for `int`, intended to be `TRUE` or `FALSE`

`intptr`: A synonym for `int*`.

`floatptr`: A synonym for `float*`.

`charptr`: A synonym for `char*`.

`voidptr`: Canopy uses `voidptr` extensively in the ANSI sense of a "pointer to anything." All of the `field_pointer` routines, for example, return `voidptr` because the `field_pointer` routine is general and does not know the type to which its return value points. Canopy uses `voidptr` in formal arguments and function return values where the argument must be a pointer to some type but that type is not fixed. Pointer arguments to Canopy routines must thus be cast as `voidptr` and return values of Canopy routines cast back as pointers to the real type, since ANSI C sometimes complains `voidptr` does not conform to a pointer to something not `void` even though this practice is officially blessed. For non-ANSI compilers lacking a `void` type, `voidptr` may be declared as either `intptr` or `charptr`, which means `*voidptr` may not be type `void`, (it may be `char` instead of `int`, for example). This can never matter for valid programs.

`intfunptr`: A pointer to a function returning an integer, declared as `typedef int (*intfunptr)()`.

`voidfunptr`: A pointer to a function with no return value, declared as `typedef void (*voidfunptr)()` or something else as appropriate for a non-ANSI compiler.

5.2 Structures

The Canopy structures define the types of variables that the user manipulates to utilize the Canopy concepts. There are also CHIP structures that the Canopy routines work with. For instance, the concept of a `full_address` is used to specify a memory location in a distributed memory system. The Canopy concept of a `site` variable is based on this underlying CHIP concept. The user manipulates `sites` and should never need to know the details of the `full_address` structure.

5.2.1 Canopy Structures

`grid`: Type for grid variables.

`field`: Type for field variables.

`set`: Type for set variables. A `grid` may be cast as a `set` and used as the set of all sites on the grid.

`map`: Type for map variables.

`site`: This type is a structure whose purpose is to point to a Canopy site. One element of this structure is a `full_address`, which can be used as an argument to ONMYNODE().

`<site>.address` The `full_address` of the "origin" of the data—including field data and other Canopy structures—for this site.

When checking whether two site variables refer to the identical site, `is_same_site()` should be used, rather than comparing `full_address`.

`path`: Structure to refer to a path along the grid. A path is an array of directions, so `path` is the same as `direction*`—a synonym for `intptr`. Note that `malloc` must be used to allocate enough space for a path—the path variable is just a pointer. An array

of integers may be cast as `path`. The path array is terminated by the special value `END`, which is an integer greater than any allowable direction.

`direction`: Type of a direction, which is an integer between `-ndir` and `+ndir` skipping zero. In `grid.h` definitions are made for the directions `X`, `Y`, `Z`, `T`, `MINUS_X`, `MINUS_Y`, `MINUS_Z` and `MINUS_T`. Since directions are small integers skipping 0, loops over directions may be done as follows:

Positive directions: `for (i=X; i<=T; i++) {}`

Negative directions: `for (i= -X; i>= -T; i--) {}`

All directions: `for (i= -T; i<=T; i=(i==-1)?1:i+1) {}`

`coordinates`: Type to store coordinates. This is a 1-based array of coordinates for each direction, with enough space to hold the largest possible number of dimensions. The 0 element is present but unused. For the rectilinear grids, the index into this array is just the direction, so that, say, `coords[X]` is the coordinate in the X direction. When a `coordinates` object is passed to a function the function should declare it as type `intptr`.

`field_address`: This type is a structure whose purpose is to point to a Canopy field element. One element of this structure is a `full_address`.

`<field_address>.fieldadd`: The `full_address` of the data composing this field element.

`sync_address`: This type is a structure with a `full_address` as its only element. Its purpose is to point to a Canopy synchronization word, and its components should only be used internally:

`<sync_address>.syncadd`: The `full_address` of the synchronization word.

5.2.2 CHIP Structures

CHIP defines structures to deal with the distributed memory nature of systems, the need for semaphores, and the control of multi-node task processing. Most Canopy programs will not use these structures directly.

Ordinary C pointers specify an address in the process memory space. On a multi-node machine a pointer must also specify the node where an address is valid. CHIP uses these structures to do so:

`full_address`: This type comprises a node number and a `voidptr`, so it may be used as a sort of "extended pointer" to point to memory anywhere in the machine. Anywhere Canopy needs to refer to a memory location that may be on an arbitrary node it uses this construct. The internal nature of a `full_addres` structure may depend on the platform on which Canopy is running. Although definitions of the elements making up this structure may be found in `chip.h`, those are not part of the public CHIP interface (and thus not listed in this manual). Explicit use of such elements may lead to non-portable code, and is deprecated.

`full_address_ptr`: Synonym for `full_address*`

CHIP has a special type for a CHIP semaphore. These provide a system-independent means of contending for resources; they are used via the routines in the SEMAPHORES section of the CHIP SUBROUTINE REFERENCE. The individual elements of the semaphore structure are not described here because they have meanings which are system dependent.

`semaphore`: A piece of memory used to keep track of a semaphore

`semaphore_ptr`: Synonym for `semaphore*`

CHIP defines the type for controlling arguments in calls to `do_task()` and `do_on_all_nodes()`. While the former is a Canopy layer concept the typedefs required for it must be defined in CHIP in order to be

passed through multi-node subroutine calls. This is the only place where a CHIP concept has been modified for Canopy.

These structures appear in the Canopy library routines establishing various sorts of do_task argument types. A user-supplied routine defining new types of integrate arguments would utilize these structures (see USER SUPPLIED ROUTINES in the subsection on DO_TASK KEYWORDS).

`CAN_do_task_keyword`: A structure describing a do_task argument. This is the type of the first argument to a do_task triplet, a structure controlling how to combine the return arguments from task routines.as described in detail in 6.4.7 TAILORING DO_TASK KEYWORDS.

`CAN_do_task_keyword_ptr`: Synonym for `CAN_do_task_keyword*`

One structure exists only as part of the host communication scheme. This is never touched by any higher layer but it must be public so the host can find it in the symbol table and interpret it:

`CHIP_node_start_frame`: Used for host communications.

5.3 Global Variables and Macros

5.3.1 Canopy Variables

Canopy makes public several global variables, which are valid on all nodes may be used inside tasks or in the control program. They may be implemented as ordinary variables or as macros (by `#define` directives). Either way, these are to be treated as *read only* variables; attempts to change their values directly will lead to undesired results.

`site *HOME`: A pointer to the `HOME` site.

`grid NOGRID`: A grid variable used as the null grid (type is `grid`).

`site NOWHERE`: The null site. The `NOWHERE` site is valid during and after the call to `complete_definitions`. Note that the variable `NOWHERE` is of type `site`, in contrast to `HOME` which is a `site*` *pointer* to a site. It is appropriate to use `NOWHERE` in user-supplied mapping functions and lattice definition functions—these functions get called by `complete_definitions`.

`site CAN_current_site_pointer`: This variable is pointed to by `HOME` and should probably not be used by itself. It is valid only inside tasks.

`int CAN_nlattices`: Number of grids.

`int CAN_nfields`: Number of fields.

`int CAN_nsets`: Number of sets.

`int CAN_nmaps`: Number of maps.

`int CAN_nrandoms`: Number of random number generators.

`int CAN_nthreads`: Maximum number of threads to be used.

`int CAN_stack_size` Size of stack assigned to each thread. The values of `CAN_stack_size` and `CAN_nthreads` may be controlled using the `multithread` routine (section 4.6.2 CONTROL OF MULTITHREADING); the user must not modify these variables directly.

`int CAN_my_thread`: Number of the thread currently executing. This is valid anywhere, but will be zero outside of tasks.

(The lattice, field, set, map and randoms counters are valid for use at all times, but before `complete_definitions()` they only count the number declared to that point.)

These lalloc structures have to be visible to the linker but should probably not be used directly:

`CAN_lalloc_structure CAN_lalloc`

`CAN_lalloc_structure CAN_control_lalloc`

`CAN_lalloc_structure CAN_do_task_lalloc`

In addition, `canopy.h` defines the following symbol for convenience in compilation:

`CANOPY_H`: This is defined to prevent errors if `canopy.h` is included more than once.

5.3.2 CHIP Variables and Macros

CHIP defines logical values (as integers), for internal use. The user is free to take advantage of these when using `logical` variables.

`TRUE`: Value 1 for logical variables.

`FALSE`: Value 0 for logical variables.

CHIP also defines the words `ZERO` through `TWENTY` as the integers 0 through 20 (in the obvious way). This can be useful when invoking routines which expect arguments passed by reference.

The following is defined as a `full_address` structure which matches no other `full_address`:

`NULL_FULL_ADDRESS`: A `full_address` that matches no other, and which is invalid to attempt to dereference.

`NODE_NUMBER(NULL_FULL_ADDRESS)` returns -1, a value which cannot match any actual node.

`local_address_from_full_address(NULL_FULL_ADDRESS)` returns NULL, a null pointer.

The `NOWHERE` site has `NULL_FULL_ADDRESS` as its `full_address`.

The following variables are established and exported by the CHIP layer routines. They are set up before the call to `control` and so are valid anywhere. The Canopy program is allowed to read them (but should not normally need to).

`int CAN_number_of_nodes`: Number of nodes used by this job.

`int CAN_number_of_this_node`: Index of this node, where the control node is 0 and the range is up to `CAN_number_of_nodes - 1`.

The following provide portable ways of extracting infromation from `full_addresses`. These can be macros in a given implementation, so user code should not attempt to pass them as a function, or to use them with pre- or post-incremented arguments:

`logical ONMYNODE(full_address *fa)`: TRUE if `*fa` is local to this node.

`logical IS_SAME_FULL_ADDRESS (full_address *fa1, full_address *fa2)`: TRUE if both arguments point to the same location in the memory space.

`int NODE_NUMBER(full_address *fa)`: Returns the node number of *fa; returns -1 if fa is NULL_FULL_ADDRESS.

`voidptr LOCAL_ADDRESS(full_address *fa)`: Returns the address (within local memory on the relevant node) of *fa; returns NULL if fa is NULL_FULL_ADDRESS.

Functions for forming `full_addresses` from node numbers and pointers are described in section 6.5.2, FULL ADDRESS FUNCTIONS.

The following macros are defined in chip.h for use by Canopy internals:

`DO_TASK_PASS`: Value for task keyword type.

`DO_TASK_FUNCTION`: Value for task keyword type.

`DO_TASK_INTEGRATE`: Value for task keyword type. All user-defined `do_task` keywords will be of this type.

`MAXARGS`: Maximum number of arguments permitted in a `do_task` call.

`MAX_NODES`: Maximum number of nodes in any given job.

`WORDSIZE`: `sizeof(voidptr)`, which in most implementations matches `sizeof(int)`.

In addition, `chip.h` defines the following symbols for convenience in compilation and inter-system compatibility:

`ANSI_PROTOTYPES`: Defined if ANSI C prototypes exist.

`BIG_ENDIAN`: Defined only if the machine is big endian.

`VARARGS`: Defined if using VARARGS

`STDARG`: Defined if using STDARGS (we are moving toward this)

`BSD`: Defined if BSD 4.2 or 4.3

`SYS_V`: Defined if SYSV Unix.

`XXX_C`: XXX is replaced by the machine currently in use—see `chip.h` for a list. This is used for language-dependent `ifdefs`. It is usually set by a `-D` option in the shell running the compilation.

`CHIP_H`: Defined after `chip.h` has been included; this prevents errors if the file is included more than once.

5.4 Keywords

5.4.1 Canopy Keywords

Canopy applications will make use of these keywords, defined as integer values in `canopy.h`, to direct the behavior of various Canopy routines.

`READ`: Keyword used by `open_field_file()`.

`WRITE`: Keyword used by `open_field_file()`.

`APPEND`: Keyword used by `open_field_file()`.

`STREAM_PER_SITE`: Keyword used to make a random number generator that keeps a separate stream of pseudo-random numbers for each site on each grid.

`STREAM_PER_NODE`: Keyword used to make a random number generator that keeps a separate stream of pseudo-random numbers only on each node, which is the minimum number of streams needed to do things in parallel.

While Canopy itself does not include `#defines` to refer to any directions, most user programs do. The file `grid.h`, for example, defines these:

`X, Y, Z, T`: Many programs define these as 1, 2, 3, and 4 for the four positive directions.

`MINUS_X, MINUS_Y, MINUS_Z, MINUS_T`: Many programs define these as −1, −2, −3, and −4 for the four negative directions. Note that with these definitions `MINUS_X = -X` and so forth.

5.4.2 Do_task Keywords

Canopy applications use keywords to describe the nature of arguments to tasks. These keywords are used by `do_task`, but also by the CHIP level routine `do_on_all_nodes` —therefore, they are CHIP concepts. These keywords are pointers to `CAN_do_task_keyword` structures which must be in the same memory location on all nodes. The DO_TASK KEYWORDS subsection of USER-SUPPLIED ROUTINES describes how such a structure is set up. The most common types of arguments are specified by the following keywords, set up by CHIP:

`PASS`: Pass any type of argument except a function.

`FUNCTION`: Pass a function.

`SUM_REAL`: Sum up the returned argument as a float.

`INTEGRATE`: Synonym for `SUM_REAL`

`MAX_REAL`: Take the maximum returned argument as a float.

`MIN_REAL`: Take the minimum returned argument as a float.

`SUM_INTEGER`: Sum up the returned argument as an integer

`MAX_INTEGER`: Take the maximum returned argument as an integer.

`MIN_INTEGER`: Take the minimum returned argument as an integer.

`SUM_DOUBLE`: Sum up the returned argument in double-precision.

`MAX_DOUBLE`: Take the maximum returned argument (double-precision).

`MIN_DOUBLE`: Take the minimum returned argument (double-precision).

`TAG_MAX_INTEGER`: Return the maximum integer value and a tag field associated with it.

`TAG_MAX_REAL`: Return the maximum float value and a tag field associated with it.

TAG_MAX_DOUBLE: Return the maximum double value and a tag field associated with it.

END: Not a CAN_do_task_keyword_ptr but rather just an integer which is used to signify the end of the do_task triplet list.

5.4.3 Other CHIP Keywords

These special values are defined in CHIP for internal use, and are not typically used by applications:

AVAILABLE: Used by semaphores.

5.5 Function Types

Canopy declares several special types of functions even though C compilers do not notice the distinction. These different declarations are made anyway since they improve clarity.

set_of_sites_func: intfunptr returning TRUE or FALSE and used to set up a set of sites.

connectivity_func: voidfunptr described in USER SUPPLIED ROUTINES and used to set up grids.

distribution_func: voidfunptr described in USER SUPPLIED ROUTINES and used to set up grids.

coordinate_func: voidfunptr described in USER SUPPLIED ROUTINES and used to set up grids.

inverse_coordinate_func: voidfunptr described in USER SUPPLIED ROUTINES and used to set up grids.

5.6 Private Canopy Types

Canopy uses an assortment of private types. These words are, of course, reserved. More information about the private types is in the `canopy.h` file.

`define_field_list`: Information about field set-up.

`define_map_list`: Information about map set-up and maps.

`lalloc_structure`: lalloc pointers.

`queue_struct`: Random number queue.

`random_generator_area`: Used by random number definition.

5.7 Canopy Limits

Several Canopy limits are set by various `#define` objects. These may be changed by changing the `#define` statement in `canopy.h`. The current value of each of these limits is in parantheses:

`MAXPARAMETERS (57)`: Maximum number of parameters in a grid definition. For periodic rectilinear grids this must be at least twice the dimension.

`MAXFIELDS (200)`: Maximum number of site fields. Each link field takes up $ndim + 1$ site fields.

`MAXLATTICES (10)`: Maximum number of grids.

`MAXSETS (200)`: Maximum number of sets. Each grid takes up one set as well as one grid.

`MAXCLUSTERS (20)`: Maximum number of field clusters. Each link field takes up one cluster.

`MAXPAIRS (20)`: Maximum number of field pairs. Each link field also takes up one pair.

`MAXFILES (5)`: Maximum number of simultaneously open field files of both the tape and disk variety.

`MAXMAPS (20)`: Maximum number of maps. Each map also takes up two fields.

`MAXGENERATORS (5)`: Maximum number of simultaneous different random number generators. Each generator also takes up at least two fields and a cluster.

`MAXTHREADS (512)`: Maximum number of threads of execution which may be active simultaneously.

Other limits are set in `chip.h`, as described in CHIP VARIABLES AND MACROS:

`MAX_NODES (630)`: Maximum number of processor nodes to be used by any one process. This may be less than the total number in the system. Implementations of Canopy on very large systems will modify this limit to allow for very large jobs.

`MAXARGS (10)`: Maximum number of argument triplets in a call to `do_task`.

`CAN_ATOMIC_GATHER (512)`: Maximum number of blocks of data that will be atomically transfered by a remote_gather or remote_scatter call.

5.8 List of All Reserved Words

This is a list of words the Canopy user is restricted from defining in an application, because the Canopy software already defines them. These words include typedefs, structs, keywords, global variables and macros,

defined in this chapter, and names of public routines, defined in the CANOPY SUBROUTINE REFERENCE chapter.

There may, depending on implementations, be other reserved words, which are inappropriate for the user to use but which need to be exported for interfacing to hosting or other tools. All such "hidden" reserved words begin with `CAN_` or `CHIP_`; every symbol is either public (and defined in this manual) or begins with one of those strings.

ANSI_PROTOTYPES
APPEND
AVAILABLE
BIG_ENDIAN
BSD
CANOPY_H
CAN_ATOMIC_GATHER
CAN_control_lalloc
CAN_current_site_pointer
CAN_do_task_keyword
CAN_do_task_keyword_ptr
CAN_do_task_lalloc
CAN_lalloc
CAN_lalloc_structure
CAN_my_thread
CAN_nfields
CAN_nlattices
CAN_nmaps
CAN_nrandoms
CAN_nthreads
CAN_nsets
CAN_number_of_nodes
CAN_number_of_this_node
charptr
CHIP_H
connectivity_func
coordinates
coordinate_func
define_field_list
define_map_list
direction
distribution_func
DO_TASK_FUNCTION
DO_TASK_INTEGRATE
DO_TASK_PASS
EIGHT
EIGHTEEN
ELEVEN
END
FALSE
field_address
FIFTEEN
FIVE
floatptr
FOUR
FOURTEEN
full_address
FUNCTION
grid
HOME
INTEGRATE
intfunptr
intptr
inverse_coordinate_func
IS_SAME_FULL_ADDRESS
LOCAL_ADDRESS
logical
map
MAXARGS
MAXCLUSTERS
MAXFIELDS
MAXFILES
MAXGENERATORS
MAXLATTICES
MAXMAPS
MAXPAIRS
MAXPARAMETERS
MAXSETS
MAX_DOUBLE
MAX_INTEGER

MAX_NODES
MAX_REAL
MINUS_T
MINUS_X
MINUS_Y
MINUS_Z
MIN_DOUBLE
MIN_INTEGER
MIN_REAL
NINE
NINETEEN
NODE_NUMBER
NOGRID
NOWHERE
ONMYNODE
ONE
PASS
path
queue_struct
random_generator_area
READ
semaphore
semaphore_ptr
set
set_of_sites_func
SEVEN
SEVENTEEN
site
SIX
SIXTEEN
stack_size
STDARG
STREAM_PER_NODE
STREAM_PER_SITE
SUM_DOUBLE
SUM_INTEGER
SUM_REAL
sync_address
SYS_V
T
TAG_MAX_DOUBLE
TAG_MAX_INTEGER
TAG_MAX_REAL
TEN
THIRTEEN
THREE
TRUE
TWELVE
TWENTY
TWO
VARARGS
void
voidfunptr
voidptr
WORDSIZE
WRITE
X
Y
Z

Chapter 6

Canopy Subroutine Reference

6.1 Declaration Routines

Before the call to `complete_definitions()` all of the grids, fields, sets, maps, and random number generators used in Canopy must be declared using these routines. The grid, set, map, and random number generator routines require user-supplied functions to determine *which* grid, set, map or random number generator is being declared. (These are detailed in the section USER-SUPPLIED ROUTINES.)

The libraries `gridlib`, `setlib`, and `ranlib` contain routines for several commonly-used constructs, which may be used as examples and templates for building customized functions for other constructs.

6.1.1 Grid Declaration

indextarbitrary_grid()

```
grid arbitrary_grid(int nsites,
                    int ndim,
                    int ndir,
                    intptr lower_limits,
                    intptr upper_limits,
                    intptr other_params,
                    distribution_func dist_func,
                    coordinate_func c_func,
                    inverse_coordinate_func icfunc,
                    connectivity_func conn_func);
```

Purpose: To declare an arbitrary grid using user-supplied functions for the grid connectivity and distribution. When `arbitrary_grid` is called, Canopy will the coordinate function and distribution function for site 1, the site 2, ... up to `nsites`,] to determine the coordinates and the node responsible for each site. Although the coordinates may have gaps, site serial numbers run from 1 to `nsites` continuously. After the coordinates and site distribution have been set up, the connectivity and inverse coordinate functions are used to create the remaining structures defining the grid.

Arguments:

`int nsites`: The number of sites in the grid.

`int ndim`: The number of dimensions of the grid.

`int ndir`: The number of positive directions of the grid. This is not necessarily the number of dimensions. For example, in a hexagonal lattice on a plane there are three positive directions but only two dimensions. Also notice that of the six possible links (in the positive and negative directions

from each site) only three are realized. This is allowed as long as the grid has the property that the site in direction $+d$ from the site in direction $-d$ from site s is site s (which simply says that the way to get back is to go in the negative direction).

`intptr lower_limits`: A one-based array containing the lower limits (or least coordinate allowed) in each dimension.

`intptr upper_limits`: A one-based array containing the upper limits (or greatest coordinate allowed) in each dimension. Notice two things: 1) The number of sites allowed in a dimension is thus `upper_limit—lower_limit + 1`; and 2) Not all the allowed coordinates need be used. For rectilinear lattices all combinations of coordinates correspond to sites. For some grids (such as hexagonal grids) this will not be so, which is fine as long as all the points have coordinates inside the limits.

`intptr other_params`: Pointer to an array which may be used to pass extra user-defined information about the grid, to various user supplied connectivity, coordinate and distribution functions (see section 6.4, `User Supplied Routines`). The number of words of data available in this manner is MAXPARAMETERS - 2*ndim. This array must always be present; if no extra data is to be used, a dummy array of dimension MAXPARAMETERS is always safe.

`distribution_func dist_func`: A pointer to a distribution function, as described in USER SUPPLIED ROUTINES.

`coordinate_func c_func`: A pointer to a coordinate function, as described in USER SUPPLIED ROUTINES.

`inverse_coordinate_func ic_func`: A pointer to an inverse coordinate function, as described in USER SUPPLIED ROUTINES.

`connectivity_func conn_func`: A pointer to a connectivity function, as described in USER SUPPLIED ROUTINES.

Return Value: The `grid` variable referring to this grid.

The following functions already exist in `gridlib` and are good examples of where to begin. The default distribution function can be used without change for any new grid, leading to sensible (though not necessarily optimal) distribution of sites among the nodes.

```
default_distribution_function
periodic_connectivity_func
periodic_coordinate_func
periodic_inv_coordinate_func
chunk_coordinate_func
chunk_inv_coordinate_func
```

6.1.2 Field Declaration

Declaration

```
field site_field(grid g, int nbytes);
field link_field(grid g, int nbytes);
```

Purpose: To declare fields on the sites or links of a previously declared grid. If the field is a site field it has one element on each site of the grid. If it is a link field it has one element on each link, and the element in direction $+d$ from site s is the *same as* the element in direction $-d$ from the site in direction $+d$ from site s.

Arguments:

`grid g`: The grid on which the new field is to be declared.

`int nbytes`: The size of each element of the new field, in bytes. Especially when a field may contain a complicated structure, it is highly recommended that `sizeof(`*fieldelement*`)` be used rather than a manual count of the size of the field. This is because many C compilers will pad a structure to allow for alignment of the data in that structure; `sizeof()` will always take this padding into account.

Return Value: The `field` variable referring to the new field.

The data reserved for fields is aligned in an appropriate manner: Fields are always at least 4-byte aligned, and `nbytes` for a field is a multiple of eight, then the field elements will be 8-byte aligned. On certain chips (the i860 in particular), this 8-byte alignment is crucial, since double-precision loads must start on an 8-byte boundary. Even if a system is capable of handling a non-aligned access, aligned accesses are more efficient.

Canopy does not support automatic quadword (16-byte) alignment.

Grouping

```
void overlap_fields(int n, field *list);
void cluster_fields(int n, field *list);
```

Purpose: To force fields to be consecutive or to share memory space. This is used primarily by the `link_field` function, which creates an overlapped and clustered field as described below.

Arguments:

`int n`: The number of fields to be overlapped or clustered.

`field *list`: An array of fields of length `n`.

Return Value: None

```
Example: A 4-D Link Field:
                /* notice that site_field returns */
                /* consecutive integers.          */
field clist[4]; /* list of clusters */
field olist[2]; /* list of overlaps */
field flink=site_field(g,4*nbytes);
clist[0]=site_field(g,nbytes);/*a field for each */
clist[1]=site_field(g,nbytes);/*direction on the */
clist[2]=site_field(g,nbytes);/*grid -- known to */
clist[3]=site_field(g,nbytes);/*be 4-dimensional */
cluster_fields(4,clist);/* force to be in order  */
olist[1] = list[0];
olist[0] = flink;
overlap_fields(2,olist);/* link field now is one */
                        /* large field overlapped*/
                        /* with four smaller ones*/
```

6.1.3 Set Declaration

```
set set_of_sites(grid g, set_of_sites_func func);
set redefine_set_of_sites(grid g,
                          set_of_sites_func func,
                          set set_to_change);
```

Purpose: To declare a set of sites for use by do_task. The redefine function is special: it is the only declaration routine that is called *after* complete_definitions time. If the `set_of_sites` function uses global variables this can be used to make a set of sites which depends on some calculation done on the grid.

Arguments:

`grid g`: The grid on which to declare the set.

`set_of_sites_func func`: The function that determines which sites are in the set and, if the set is compound, their level in the set. This function is described in the USER SUPPLIED ROUTINES section.

`set set_to_change`: For a redefinition, this is the set variable of the set to be changed. This set must have been already declared on the same grid as the new definition.

Return Value: The newly declared `set`. For redefinitions, this is the same as set_to_change.

6.1.4 Map Declaration

```
map define_map(grid domain,
               grid range,
               intfunptr mapfunc);
map compose_map(map mid_to_range,
                map domain_to_mid);
```

Purpose: To declare maps. `define_map` declares a map from a function; `compose_map` declares the composition of two maps. The maps in a composition must have already been declared and the range of the second must be the domain of the first, which is required by the mathematical definition of composition. Maps may be automorphisms, and there may be more than one map connecting the same grids.

Arguments:

`grid domain`: The domain grid for the new map.

`grid range`: The range grid for the new map.

`intfunptr mapfunc`: A pointer to the mapping function, as described in USER SUPPLIED ROUTINES. Note that `mapfunc` returns a `logical` rather than an integer, but that `logical` is a synonym for the subset (0,1) of the integers so the types really do conform.

`map mid_to_range`: The map applied second in a composition.

`map domain_to_mid`: The map applied first in a composition. The composed map maps a site in the domain grid of the `domain_to_mid` map to the range grid of the `mid_to_range` map. The site maps to `NOWHERE` in the obvious cases.

Return Value: The newly declared map.

6.1.5 Random Number Declaration

```
int make_random_generator(voidfunptr random_func,
                          int type,
                          int number_to_make,
                          int seed);
```

Purpose: To declare a random number generator.

Arguments:

`voidfunptr random_func`: The function returning pointers to the generation and initialization functions, as described in USER SUPPLIED ROUTINES.

`int type`: Either `STREAM_PER_SITE` or `STREAM_PER_NODE`.

`int number_to_make`: The number of random numbers to be generated in a single call to the generation function. It is more efficient to use larger values for this number but it takes more storage, particularly in the stream-per-site case. The random sequence does not depend on this number.

`int seed`: The seed value. The same seed value leads to the same sequence.

Return Value: The integer number of the random number generator as used by `multi_random`.

Note: In stream-per-site mode each site on each grid is assigned a unique *stream number* from its coordinates and grid number. Therefore changing from, say, a periodic grid to a chunky periodic grid of the same size has no effect on the random numbers, but changing the program so a new grid is declared before the old grid will change the streams. With a little thought programs may be revised so the random number sequence is the same if the same seed is used—this simplifies debugging.

6.1.6 Complete_Definitions

```
void complete_definitions();
void complete_canopy_handshake();
```

Purpose: `complete_definitions` creates the internal structures used to control all of the declared objects. All of the declarations except `redefine_set_of_sites` must precede it, and only one call to it may be made in a program. A call to `complete_definitions` also causes the linker to load in all of the Canopy structure that calls the `control` entry point, so a C program may be converted to a Canopy program by adding the line `complete_definitions` and changing the main entry point from `main` to `control`.

`complete_canopy_handshake` may be used instead of `complete_definitions` to run a prgram with *no* other Canopy library routines, on a platform which expects Canopy applications. This makes the executable image smaller, since the internal Canopy routines called by `complete_definitions` need not be linked in.

Arguments: None

Return Value: None

6.2 Routines Called By Control Program

6.2.1 Do_Task Routines

Tasks Using Sets

```
void do_task(voidfunptr task, set s, ..., END);
void do_task_n_times(voidfunptr task, set s,
                     int ntimes, ..., END);
```

Purpose: To call some subroutine on each site in a set. This is how Canopy effects parallelism: The sites in the set are done on individual nodes in parallel. An example of how to use this is given later in this section: DO_TASK EXAMPLE.

Arguments:

`voidfunptr task`: The task to be done on set `s`. Task functions are described in detail in USER SUPPLIED ROUTINES.

`set s`: The set on which to do the task. If `s` is a compound set later levels are done only after earlier levels are completed, but consult the description of compound set usage in the PARALLELISM CONSIDERATIONS of the TUTORIAL chapter for details of how to ensure synchronization.

`int ntimes`: For `do_task_n_times`, this is the number of times to do the task. This is like creating a compound set of sites with each site in the set several times, so the synchronization routines must be used the same way.

`...`: Triplets of arguments, as described in DO_TASK TRIPLETS.

`END`: The keyword to end the variable argument list.

Return Value: None

Sub-Tasks Using Maps

```
void do_task_on_inverse_image(voidfunptr task, map m,
                              ..., END);
void do_task_on_inverse_image_set
                    (voidfunptr task, map m, set s,
                     ..., END);
```

Purpose: To do a sub-task. These must be called inside another task routine. They do a task on the inverse image of the HOME site. These calls may be nested to any level, as the HOME site in the sub-task may have another inverse image by another map.

Arguments:

`voidfunptr task`: The task to be done on set `s`. This task function is described in detail in USER SUPPLIED ROUTINES.

`map m`: A map whose *range* grid must be the grid of the current HOME site. The task is done on those sites on the *domain* grid of `m` in the inverse image of the HOME site.

`set s`: A set in the *domain* of the map used to restrict the inverse image. The task will be done *only* on those sites both in the inverse image of the HOME site and in set `s`. If `s` is a compound set, the sites will be done in the order described by the levels just as in `do_task` itself.

`...`: Triplets of arguments, as described in DO_TASK TRIPLETS.

`END`: The keyword to end the variable argument list.

Return Value: None

A warning: During a sub-task, the fields on the HOME site are *not* guaranteed to reside on the local node. During sub-tasks, `put_field` must be used to modify field data, even at the HOME site.

Do_Task Triplets

Ordinary subroutines only need to know the *address* or *value* of their arguments. Because Canopy task subroutines are done over a set of sites on (possibly) different CPU's, they need to also know the *length* of their arguments. In addition, Canopy tasks need to know what to do with output arguments from a task on a site, since there is only one output returned to the calling routine instead of one for each site. This is controlled by the *keyword* governing the argument. For internal reasons, arguments to task routines must all be *pass by address*. A complete triplet is of the form:

`<keyword>, <address of argument>, <length in bytes>`

Several *keyword* arguments have already been defined:

`FUNCTION`: To pass a function address.

`PASS`: To pass any other variable.

`INTEGRATE`: To sum up the arguments after the task routine returns, treating them as real numbers.

`SUM_REAL`: Synonym for `INTEGRATE`

`MAX_REAL`: To keep only the maximum argument, treating them as real numbers.

`MIN_REAL`: To keep only the minimum argument, treating them as real numbers.

`SUM_INTEGER`: To sum up the arguments, treating them as integers.

`MAX_INTEGER`: To keep only the maximum argument, treating them as integers.

`MIN_INTEGER`: To keep only the minimum argument, treating them as integers.

`SUM_DOUBLE`: Double-precision sum.

`MAX_DOUBLE`: Double-precision maximum.

`MIN_DOUBLE`: Double-precision minimum.

`TAG_MAX_INTEGER`: Used to return a tag field. Returns the maximum value of the first element of a structure (treated as an integer) and the remainder of the structure associated with the maximum value. For example, this could be used to return the coordinates of the site with the largest value.

`TAG_MAX_REAL`: As above, but treats the first element as a float.

`TAG_MAX_DOUBLE`: As above, but treats the first element as a double.

Obviously, `PASS` and `FUNCTION` are used when the task routine is expecting a pointer to an input argument, and the others are used when the task routine computes a value and stores it. The length of the arguments is also needed so that `do_task` can process the return values properly. Users may create additional keywords as described in the section on USER-SUPPLIED ROUTINES.

Examples of triplets are:

```
PASS, &this, sizeof(this),
INTEGRATE, &somefloat, sizeof(float),
MAX_INTEGER, &intarray, arraydimension*sizeof(int),
MIN_REAL, &realval, sizeof(realval),
FUNCTION, functionname, sizeof(functionname),
```

For the `FUNCTION` keyword the argument is always a function name and the length is always the size of a pointer (function pointers have the same length as any other pointer type). If an array of function pointers is passed to the task routine (which is completely legal since functions are guaranteed to have the same address on all nodes) use `PASS` instead.

On the following page is an example of how to use tag fields to write a routine that returns the coordinates of the site with the largest value of some field.

```
typedef struct {  /* Define the structure first */
  float value;
  coordinates coords;
} test_tag;

void task_(tt,f)            /* the task routine */
test_tag *tt;
field *f;
{
          /* the value and its coordinates */
  tt->value = *((float*)field_pointer(f,HOME));
  get_coordinates(HOME, tt->coords);
} /* task_ */

void task(f)/* this routine for 2-D grids  only */
field f;
{
  test_tag tt; /* temporary */
  do_task(task_, grid_supporting_field(f),
          TAG_MAX_REAL, &tt, sizeof(tt),
          PASS, &f, sizeof(f),
          END);
  printf("%f at (%d,%d)\n",tt.value,
             tt.coords[X],tt.coords[Y]);
} /* task */
```

Do_Task Example

```
/* Here is an example of do_task using the      */
/* 'generalized subroutine header' structure.   */

       /* test_task_ returns the sum of the co- */
       /* ordinates at a site multiplied by an  */
       /* integer in the array inarg.           */
       /* The output is a float.                */
void test_task_(inarg, outarg);/* note _ at end */
int *inarg;
float *outarg;  /* everything is pass-by-address*/
{
  coordinates coords;
  grid g = grid_supporting_site(HOME);
  int ndim = number_of_dimensions_of_grid(g);
  int i;
  get_coordinates(HOME,coords);
  for (*outarg=0.0, i=1; i<=ndim; i++) {
    *outarg += inarg[i]*coords[i];
  }
} /* test_task_ */

void test_task(s, inarg, outarg) /* routine to   */
set s;        /* call test_task_ through do_task */
int *inarg;
float *outarg;
{
  grid g = grid_supporting_set(s);
  int ndim = number_of_dimensions_of_grid(g);
  do_task(test_task_, s,
          PASS, inarg, ndim*sizeof(int),
          INTEGRATE, outarg, sizeof(float),
          END);
} /* test_task */
```

```
                    /* this is a sample control  */
                    /* program using test_task.  */
void control()
{
  grid g;
  coordinates a;
  float outval;
                    /* define the grid here      */
  g = periodic_cubic_grid(3,4,5);
                    /* This call must be in all  */
                    /* Canopy programs after the */
                    /* declarations are done.    */
  complete_definitions();

                    /* set up the array for inarg*/
  a[1] = 1; a[2] = 2; a[3] = 4;
  test_task((set) g, a, &outval);

                    /* print the result          */
  printf("And the answer is %f\n",outval);
  printf("And the answer should be 270.0\n");
} /* control */
```

6.2.2 Broadcast

```
void broadcast(voidptr object, int length);
```

Purpose: To make the value of a global variable set in the `control` program known to all of the task routines on every other node as well.

Arguments:

`voidptr object`: The address of the static variable. This address is cast as a `voidptr` since Canopy uses that type as a "pointer to anything."

`int length`: The length in bytes of the object to be broadcast.

Return Value: None

NOTE: This cannot be called from inside a task routine, nor can it be used on variables obtained through dynamic allocation (since those variables will not be in the same place on all nodes). It also makes a temporary copy of the object being `broadcast`, so very large objects should be broken into pieces if `malloc` runs out of space.

6.2.3 Field File Routines

Opening and Closing Files

```
void open_field_file(char *filename, int rwmode);
void close_field_file(char *filename);
```

Purpose: To open and close Canopy field files. These must not be called from a task routine.

Arguments:

`char *filename`: The field file name. If the name does not contain a '#' sign, the file is an ordinary UNIX file on the host machine. If the name does contain a '#' then the field file is stored on the distributed file system. The format for file-names on the distributed system is `setname#filename`, where `setname` identifies the tape set or (`disk`*nn*) the multi-disk set. Tape sets must have been already initialized and mounted using the appropriate tape system commands before `open` is called. See the User's Guide for more details about mounting tapes and about set names for files on disk.

`int rwmode`: This is the keyword value `READ` or `WRITE`, depending on whether the field file is to be opened for read, write. On the distributed file system, open for write will only create a new file—if a file already exists open will fail. For disk files, a third option is `APPEND`, which allows new field records to be appended to existing files (or creates a new file if `filename` does not exist).

Return Value: None. Errors in file open or close operations are reported as fatal errors.

Writing and Reading Fields

```
void write_field(char *filename, field f);
void read_field(char *filename, field f);
void read_slice_of_field
       (char *filename, field f, intfunptr mapfunc);
```

Purpose: To read and write fields on field files. (These must not be called from a task routine.) Use `read_slice_of_field()` to input a subset of field elements from a stored field. For example, one time-slice of a 4-dimensional grid may be input onto a field defined on a 3-dimensional grid.

Arguments:

`char *filename`: The field file name as used in the `open_field_file()` call.

`field f`: The field to read or write: `read_field` and `write_field` read or write the entire field to disk or tape.

`intfunptr mapfunc`: For `read_slice_of_field`, this is a pointer to the mapping function specifying where each input field element is to be placed. See the section Mapping Functions in USER SUPPLIED ROUTINES for a description of the function syntax. The input coordinates of the map refer to the grid associated with the field stored on disk or tape; the output coordinates are on the grid supporting the field `f`. For given input coordinates, if `mapfunc` returns FALSE, then the field element at that site is not read in at all.

Return Value: None. Errors are fatal.

6.2.4 IEEE Precision Control

On those machines which have IEEE floating-point arithmetic (such as the D860 and FPAP nodes) these routines provide a unified method of setting the mode in use. Canopy programs print out the IEEE mode they are using at the beginning so there is no confusion. Not all CPU's support all modes and the defaults may be different for different machines.

The modes and flag names are described on the next page. Canopy reads the enviroment variable FMODE to over-ride the default IEEE modes. For example,

```
setenv FMODE="ROUND_UP,UNDERFLOW_ZERO"
```

selects round toward plus infinity and set underflows to zero. The default modes for some existing target machines are:

FPAP: Round to nearest, underflow to zero, all the rest IEEE.

D860: Round to nearest, underflow to zero, all the rest abort.

ULTRIX: IEEE handling unavailable.

SGI: Round to nearest, all the rest IEEE.

The reason the defaults are different is that the IEEE modes are not provided on all platforms. As the system matures full IEEE handling will be available for D860's eventually.

Rounding Mode: Sets the IEEE rounding mode

FMODE_ROUND_TO_NEAREST: The usual IEEE handling whereby several guard bits are kept.

FMODE_ROUND_UP: Round toward positive infinity.

FMODE_ROUND_DOWN: Round toward negative infinity.

FMODE_ROUND_TO_ZERO: Round toward zero.

Underflow Handling: Specifies underflow handling

FMODE_UNDERFLOW_IEEE: Does IEEE "soft" underflow by using denormals to represent numbers smaller than the least possible floating-point value available with full precision.

FMODE_UNDERFLOW_ZERO: Sets the result to zero when an underflow occurs.

FMODE_UNDERFLOW_ABORT: Stops the job if an underflow occurs.

Overflow Handling: Specifies overflow handling

FMODE_OVERFLOW_IEEE: Sets the result to +Inf or -Inf if an overflow occurs, depending on the sign of the result.

FMODE_OVERFLOW_ABORT: Stops the job if an overflow occurs.

Zero Divide Handling: Specifies handling of divide-by-zero errors.

FMODE_ZERO_DIVIDE_IEEE: Sets the result to +Inf or -Inf if a zero-divide error occurs depending on whether the dividend was positive or negative.

FMODE_ZERO_DIVIDE_ABORT: Stops the job if a zero-divide occurs.

Invalid Handling: Specifies handling of illegal subroutine arguments.

FMODE_INVALID_IEEE: Sets the result to the IEEE specified value if an invalid operand is detected.

FMODE_INVALID_ABORT: Stops the job if an invalid operand is detected.

```
int set_default_floating_mode();
int set_floating_mode_to_environment();
```

Purpose:

Arguments: None

Return Value: If there were no errors these return 0, if there are errors (which cannot occur setting the default mode) the return is -1 and the reason for failure is printed on `stderr`.

```
int get_current_floating_mode();
```

Purpose: To return the currently set floating-point mode. This is the logical or of all the currently set mode flags.

Arguments: None

Return Value: The currently set floating modes.

```
void print_current_floating_mode();
```

Purpose: Prints the currently set floating-point mode on `stdout` in human-readable form.

Arguments: None

Return Value: None

```
int set_current_floating_mode(int flags);
```

Purpose:

Arguments:

`int flags`: The flag bits for the desired mode, or'ed together. A full specification would be five flag bits set, one for each of the exception types. If no bit is specified for an exception, the mode for that exception is unchanged. For example, setting flags to `FMODE_UNDERFLOW_ZERO` has no effect on the rounding, overflow, zero divide, or invalid handling.

Return Value: If there were no errors these return 0, if there are errors (which cannot occur setting the default mode) the return is -1 and the reason for failure is printed on `stderr`.

6.2.5 Transfer Coalescing

These routines control the efforts to minimize he number of actual data transfers by coalescing accesses required by `field_pointer` and `put_field` routines. For some systems (including any single-CPU implementation of Canopy), multi-threading and transfer coalescing are null concepts or unnecessary. These routines are present (but have not effect) in Canopy on such systems.

Multi-thread Setup

```
void multithread(int nthreads, int stack_size);
```

Purpose: To establish multi-thread operation, in which some number of threads may be active at one time on each node, to allow for coalescing of multiple transfers to other remote nodes. An explicit call to multithread() will override any multi-thread mode settings established in other ways. This function can be called multiple times in an application to dynamiccally alter the number fo threads or stack size; specifying nthreads=1 will disable multi-thread mode. Note that calling multithread() is a relatively time-consuming operation.

Arguments:

`int nthreads`: The maximum number of threads that will be active simultaneously on each node.

`int stack_size`: The amount of local stack space reserved for each potential thread. If multi-threading is enabled from the command line, this defaults to a reasonable value (8K bytes).

Return Value: None.

Multi-thread Control

```
void multithread_disable ();
void multithread_enable ();
```

Purpose: To temporarily disable (and later enable) multi-thread mode. This may be desirable under conditions discussed in section 4.6.2 CONTROL OF MULTI-THREADING. A task cannot enable multi-thread mode if it has been disabled by its "parent": If the control program disables multi-thread, then `multithread_enable()` has no effect inside a task routine, and if a task disables multi-thread, then it cannot be enabled within any subtask it invokes. Unlike multithread(), these routines are not time-consuming.

Arguments: None.

Return Value: None.

```
void print_multithread_stats ();
```

Purpose: To print statistics which may be useful for fine-tuning the number of threads and/or stack size for an application. Information reprorted includes the average degree of coalescing and stack usage statistics. The statistics are reset each time `print_multithread_stats()` is called. This function may not be called from inside a task routine.

Arguments: None.

Return Value: None.

6.3 Routines Called During Tasks

This section describes routines for manipulating Canopy concepts. These routines in this section are typically called by task routines, but may also invoked by the control program.

6.3.1 Site Manipulation

Absolute Site Location

```
site site_at_coordinates(grid g, intptr coords);
```

Purpose: To set a `site` variable to the location specified by `coordinates` on grid `g`.

Arguments:

`grid g`: The grid on which to place the site.

`intptr coords`: The coordinates to place the site, in a one-based `coordinates` array (see GETTING COORDINATES in section 6.3.7).

Return Value: A `site` variable at the desired location.

```
Examples:
coordinates coords;
site s1, s2, s3;
coords[X] = 1; /* X, Y, Z, T are 1, 2, 3, 4 */
coords[Y] = 3;
coords[Z] = 0;
coords[T] = 4;
s1 = *HOME; /* HOME is a pointer a site! */
s2 = site_at_coordinates(g, coords);
s3 = s2;            /* set s3 equal to s2 */
```

Relative Site Location

```
site move_site(site *startsite, direction dir);
site move_site_by_path(site *startsite, path p);
site site_at_dir(direction dir);
site site_at_path(path p);
```

Purpose: To return a site variable offset from another site variable or the HOME site. The move_site, move_site_by_path, and site_at_path routines are illegal while vertical coalescing is active.

Arguments:

site *startsite: A pointer to a site.

direction dir: A direction used to specify the site in direction dir from *startsite or HOME. If dir is zero, then the site returned is startsite (or HOME for site_at_dir).

path p: A path used to specify the site at the end of path p from *startsite or HOME.

Return Value: A site variable at the desired location.

Note: Inside a task routine HOME may be used as a pointer to a site, so the site_at_dir and site_at_path routines can be built out of the others:

```
move_site(HOME, dir) <--> site_at_dir(dir)

move_site_by_path(HOME, p) <-->  site_at_path(p)
```

Site Comparison

```
logical is_same_site(site *s1, site *s2);
```

Purpose: To test if two sites are the same.

Arguments:

`site *s1`: A pointer to a site.

`site *s2`: A pointer to another site.

Return Value: `TRUE` if `*s1` and `*s2` are the same, `FALSE` if not.

Note: The special site `NOWHERE` matches only itself.

Site Information

```
grid grid_supporting_site(site *s);
```

Purpose: To return the grid on which a site is located.

Arguments:

`site *s`: A pointer to a site.

Return Value: The grid where `*s` lives. If the argument is `NOWHERE` then the return is `NOGRID`. This can create confusion with maps, since it means that the grid supporting the site of a mapped image may be `NOGRID` and not the range grid of the map.

Mapping Sites

```
site image_of_site(map m, site *s);
site *inverse_image_of_site(map m, site *s);
```

Purpose: To use a map to find the image and inverse image of a site.

Arguments:

`map m`: The map to use for mapping `*s`.

`site *s`: A site on the domain of `m` for `image_of_site` and on the range of `m` for `inverse_image_of_site`.

Return Values: For `image_of_site`, the return is the site on the range of `m` mapped to by `*s`. If `*s` does not map to any site in the range under `m`, the function returns the special site `NOWHERE`.

For `inverse_image_of_site`, the return is a pointer to a list of sites in the inverse image terminated by `NOWHERE`. This list is not guaranteed to be in any particular order but it does include all of the sites which map to `*s` under `m`. If no sites map to `*s`, the first element in the list is `NOWHERE` (the return is *not* a null pointer).

6.3.2 Field Manipulation

Field_Pointer Routines

```
voidptr field_pointer(field f, site *s);
voidptr field_pointer_at_dir(field f, direction dir);
voidptr field_pointer_at_path(field f, path p);
```

Purpose: To return a pointer to a read-only copy of the field element at the desired site. If the site is HOME inside a task (but not a sub-task) then the pointer points to the *actual* field element and not to a copy. Except in that case, modifying the copy of the field element is not permitted.

Arguments:

`field f:` This is the `field` variable whose value at some site is desired. If `field` was a `link_field` this routine returns a pointer to the cluster of links in all the positive directions.

`site *s:` Specifies the desired site absolutely. May be HOME.

`direction dir:` Specifies the site in direction `dir` from the HOME site.

`path p:` Specifies the site at the end of path `p` from the HOME site.

Return Value: A pointer to a *copy* of the field element at the specified site. (See the caveat in PURPOSE above.) While this is returned as a `voidptr`, it is used as a pointer to whatever type the field element actually is. Hence the typical call looks like this:

```
whatsit *w;
w = (whatsit*) field_pointer(whatsitfield,HOME)
```

Put_Field Routines

```
void put_field(field f, site *s, voidptr object);
void put_field_at_dir
        (field f, direction dir, voidptr object);
void put_field_at_path
        (field f, path p, voidptr object);
```

Purpose: To copy `object` into the specified field element. Except during sub-tasks, this is normally only used to store objects not on the `HOME` site, since `field_pointer` returns a pointer directly to the field element on the `HOME` site.

Arguments:

`field f`: The `field` variable of the desired site field.

`site *s`: Specifies the desired site absolutely. May be `HOME`.

`direction dir`: Specifies the site in direction `dir` from the `HOME` site.

`path p`: Specifies the site at the end of path `p` from the `HOME` site.

`voidptr object`: Pointer to the data to be copied into the field element. This is a `voidptr` because it must accept pointers to all types of objects. This argument is usually cast explicitly to `voidptr` in the call.

Return Value: none.

Link_Field_Pointer Routines

```
voidptr link_field_pointer
        (field f, direction link, site *s);
voidptr link_field_pointer_at_dir
        (field f, direction link, direction dir);
voidptr link_field_pointer_at_path
        (field f, direction link, path p);
```

Purpose: To return a pointer to the element of the link field `f` on the link in direction `link` emanating from the specified site. If the site is `HOME` inside a task (but not a sub-task) *and the direction is positive*, then the pointer points to the *actual* field element and not to a copy. Except in that case, modifying the copy of the field element is not permitted.

Arguments:

`field f`: The `field` variable of the desired link field.

`direction link`: The desired link direction. This may be positive or negative, but if it is negative then remember that it is the same as the link in the positive direction from the site at its other end.

`site *s`: Specifies the desired site absolutely. May be `HOME`.

`direction dir`: Specifies the site in direction `dir` from the `HOME` site.

`path p`: Specifies the site at the end of path `p` from the `HOME` site.

Return Value: Pointer to a *copy* of the link field element. (See the caveat in PURPOSE above.)

Put_Link_Field Routines

```
void put_link_field
   (field f, direction link, site *s, voidptr object);
void put_link_field_at_dir
   (field f, direction link, direction dir,
                                        voidptr object);
void put_link_field_at_path
   (field f, direction link, path p, voidptr object);
```

Purpose: To copy `object` into the element of link field `f` at the specified site. This is normally only used to store objects on links not in positive directions from the HOME site, since `link_field_pointer` returns a pointer directly to link field elements on links in positive directions from the HOME site.

Arguments:

`field f`: The `field` variable of the desired link field.

`direction link`: The desired link direction. This may be positive or negative, but if it is negative then remember that it is the same as the link in the positive direction from the site at its other end.

`site *s`: Specifies the desired site absolutely. May be HOME.

`direction dir`: Specifies the site in direction `dir` from the HOME site.

`path p`: Specifies the site at the end of path `p` from the HOME site.

`voidptr object`: Pointer to the data to be copied into the field element. Normally this must be cast as a `voidptr`.

Return Value: none.

6.3.3 Field Manipulation For Compound Tasks

Synchronization Routines

```
void synchronize(site *s);
void synchronize_at_dir(direction dir);
void synchronize_at_path(path p);
```

Purpose: To wait for a site to reach the current synchronization level, which is needed only when using compound sets of sites. The machine waits until the target site has been processed by this call to `do_task`. The intended use of these routines is to ensure, in compound tasks, that each site whose level is less than the current level has indeed completed before its field is read. For `do_task_n_times`, the synchronization routines ensure that the previous sweep has been completed on all the needed sites before their field elements are read.

Arguments:

`site *s`: Specifies the desired site absolutely. May be `HOME`.

`direction dir`: Specifies the site in direction `dir` from the `HOME` site.

`path p`: Specifies the site at the end of path `p` from the `HOME` site.

Return Value: none.

```
sync_address sync_word(site *s);
sync_address sync_word_at_dir(direction dir);
sync_address sync_word_at_path(path p);
```

Purpose: Returns the sync_address for the desired site. This is used by synchronize_with_sync_word to wait for a site to reach a certain level in a compound task.

Arguments:

site *s: Specifies the desired site absolutely. May be HOME.

direction dir: Specifies the site in direction dir from the HOME site.

path p: Specifies the site at the end of path p from the HOME site.

Return Value: The sync_word for the desired site.

```
void synchronize_with_sync_word(sync_address *sync);
```

Purpose: Same as synchronize, but uses a previously computed sync instead of a site.

Arguments:

sync_address *sync: The previously computed sync_address.

Return Value: none

Synchronize and Get Pointer Routines

```
voidptr sync_field_pointer
        (field f, site *s);
voidptr sync_field_pointer_at_dir
        (field f, direction dir);
voidptr sync_field_pointer_at_path
        (field f, path p);
```

Purpose: These are optimized combinations of `synchronize` and `field_pointer`.

Arguments:

`field f`: The `field` variable of the desired site field.

`site *s`: Specifies the desired site absolutely. May be `HOME`.

`direction dir`: Specifies the site in direction `dir` from the `HOME` site.

`path p`: Specifies the site at the end of path `p` from the `HOME` site.

Return Value: Pointer to a copy of the field element.

6.3.4 Direct Field Addressing

Field Address Creation

```
field_address address_of_field
        (field f, site *s);
field_address address_of_field_at_dir
        (field f, direction dir);
field_address address_of_field_at_path
        (field f, path p);
field_address address_of_link_field
        (field f, direction link, site *s);
field_address address_of_link_field_at_dir
        (field f, direction link, direction dir);
field_address address_of_link_field_at_path
        (field f, direction link, path p);
```

Purpose: To pre-compute a field_address variable for later use. This saves only a little time.

Arguments:

`field f`: The `field` variable of the desired site or link field.

`direction link`: The desired link direction. This may be positive or negative, but if it is negative then remember that it is the same as the link in the positive direction from the site at its other end.

`site *s`: Specifies the desired site absolutely. May be `HOME`.

`direction dir`: Specifies the site in direction `dir` from the `HOME` site.

`path p`: Specifies the site at the end of path `p` from the `HOME` site.

Return Value: The `field_address` for later use.

Field Address Use

```
voidptr field_pointer_from_address
        (field_address *where);
voidptr sync_field_pointer_from_address
        (field_address *where, sync_address *sync);
void put_field_at_field_address
        (field_address *where, voidptr object);
int length_of_field_address_field
        (field_address *where);
```

Purpose: To use precomputed `field_address` variables instead of doing the computation each time. Otherwise these routines behave like their brothers. Note that this approach saves only a little time.

Arguments:

`field_address *where`: The previously computed field address.

`sync_address *sync`: The previously computed sync address.

`voidptr object`: Pointer to the data to be copied into the field.

Return Values: Same as for the `field_pointer` routines.

6.3.5 Path Manipulation

```
int make_path(path p, ...);
int extend_path(path p, direction dir);
int concat_path(path dest, path source);
int copy_path(path dest, path source);
int path_length(path p);
```

Purpose: These routines create, extend, concatenate, and copy paths.

Arguments:

`path p`: A path variable (that is, an intptr) pointing to at least as many available words as `1+path_length`.

`...`: A list of directions terminated by `END`.

`direction dir`: A direction to add to the end of the path.

`path dest`: The target of a concatenation or copy.

`path source`: The source of a concatenation or copy.

Return Value: All of these routines return the length of the new path.

Example:

```
direction *p1, *p2;
p1 = (direction *)malloc(10*sizeof(direction));
p2 = (direction *)malloc(10*sizeof(direction));
make_path((path)p1, X, Y, END);       /* returns 2 */
make_path((path)p2, -X, -Y, END);     /* returns 2 */
concat_path((path)p1,(path)p2);       /* returns 4 */
path_length((path)p1);                /* returns 4 */
```

Manipulating Paths Directly

A path variable is a pointer to a direction. The path is an array of directions in order, terminated by the special value END, which is defined in canopy.h as an integer out of the range permissible for genuine directions. At the end of the previous example, path *p1* would be

```
X, Y, -X, -Y, END
```

Any ordered, END-terminated array of directions may be used wherever a Canopy routine expects a path argument. It is legal (and sometimes convenient) to deal with the direction* path variable directly. For example, to create a null-terminated array of paths initialized to values for subsequent use:

```
direction pathx[] = {T, T, X, X, -T, -T, END};
direction pathy[] = {T, T, Y, Y, -T, -T, END};
direction pathz[] = {T, T, Z, Z, -T, -T, END};
direction longpath[] = {X, Y, Z, T, -X, -Y, -Z, T, END};

path mypatharray[]= {pathx,pathy,pathz,longpath,NULL};
```

In this example, a path is initialized by declaring it as an array of directions. C syntax does not support the following slightly cleaner construct:

```
path pathx = {T,T,X,X,-T,-T,END}; /* won't compile! */
```

6.3.6 Informative Routines

Information About Grids

```
intptr grid_lower_bounds(grid g);
intptr grid_upper_bounds(grid g);
intptr grid_parameters(grid g);
int number_of_directions_of_grid(grid g);
int number_of_dimensions_of_grid(grid g);
int number_of_fields_on_grid(grid g);
int number_of_sets_on_grid(grid g);
```

Purpose: To return the desired data about the grid at any time after it has been defined.

Arguments:

`grid g`: The grid for which information is desired.

Return Values: The first three routines return pointers to read-only arrays. For the *supplied* grids, `grid_parameters` is meaningless. The `grid_upper_bounds` and `grid_lower_bounds` routines both return a pointer to an array of integers of length grid-dimensions containing the lowest and highest coordinate in each dimension. Note that the return value points to a 1-based array of integers (the same as the `coordinates` type) and it is illegal to use the 0 element.

```
intptr lower = grid_lower_bounds(g);
intptr upper = grid_upper_bounds(g);
```

For the supplied grids, `lower[X]`, `lower[Y]`, ... are now all 0 and `upper[X]` is the maximum coordinate in the `X` direction (which is 7 if the lattice size is 8).

Information About Fields

```
grid grid_supporting_field(field f);
int field_length(field f);
```

Purpose: To return data about a `field`.

Arguments:

`field f`: The field for which information is desired.

Return Values: The grid on which field `f` is defined and the length in bytes of an element of field `f`.

Information About Sets

```
grid grid_supporting_set(set s);
int number_of_sites_in_set(set s);
int number_of_levels_in_set(set s);
intptr nsites_at_each_level(set s);
int level_of_site_in_set(set s, site *ss);
```

Purpose: To return data about a set. Information about the number of sites is available only after the call to `complete_definitions`.

Arguments:

`set s`: The set for which information is desired. Note that a grid may be used as the set of all sites on that grid if it is cast as a set.

`site *ss`: For `level_of_site_in_set` this is the site whose level is desired.

Return Values: The grid on which set `s` is defined; the total number of sites in set `s`; the number of levels in set `s` (which is 1 if `s`: is not a compound set); a pointer to a read-only array whose **0** element is the total number of sites in set `s` and whose **level** element is the number of sites at that level. For non-compound sets, the number of sites at level 1 is the same as the total number of sites in the set. For `level_of_site_in_set`, the site's level. If the site is not in the set it returns zero; if the site is not on the same grid as the set a fatal error is declared.

Information About Maps

```
grid domain_grid_of_map(map m);
grid range_grid_of_map(map m);
map *maps_connecting_grids(grid domain, grid range);
logical is_same_map(map m1, map m2);
```

Purpose: To return information about maps. Note that `is_same_map` can only be called from the control program.

Arguments:

`map m`: The map for which information is desired

`grid domain`: domain grid to check for maps

`grid range`: range grid to check for maps

`map m1`: One of two maps to test for sameness

`map m2`: One of two maps to test for sameness

Return Values: `maps_connecting_grids` returns a pointer to a list of those previously declared maps with the specified domain and range. This list is terminated by `(map)0` (a null pointer). The `is_same_map` routine returns `TRUE` if `m1` and `m2` are isomorphic and `FALSE` otherwise.

6.3.7 Using Coordinates

Getting Coordinates

```
void get_coordinates(site *s, intptr coords);
void get_coordinates_at_dir
              (direction dir, intptr coords);
void get_coordinates_at_path
                    (path p, intptr coords);
```

Purpose: To copy the coordinates of a site into an array.

Arguments:

`intptr coords`: The array where the coordinates will be placed. Note that coordinates range from `1` to `ndim`. This means that X, Y, ... are the indices into this array.

`site *s`: A pointer to the site.

`direction dir`: Specifies the site in positive or negative direction from the `HOME` site.

`path p`: Specifies the site at the end of a path from the `HOME` site.

Return Value: None.

```
Example
coordinates mycoords;
get_coordinates(HOME,mycoords);
/* now mycoords[X] ... mycoords[Z] make sense */
```

Printing Coordinates

```
void sprintf_site_coordinates(char *ss, site *s);
```

Purpose: To print the coordinates of site `*s` in the string `*ss`. The coordinates will be formatted using "`(%d, %d, ...)`". This routine is intended for use by the `control` program to assist in printing out site locations.

Arguments:

`char *ss`: A pointer to the string where the coordinates will be `sprinted`.

`site *s`: A pointer to the site.

Return Value: None.

6.3.8 Obtaining Random Numbers

```
float random();
float multi_random(int generator);
```

Purpose: To return the next pseudo-random number in a sequence set up by the `make_random_generator` declaration.

Arguments:

`int generator`: If more than one random number generator was declared, `multi_random` uses this value to select which one to use. The `random` function returns a random number using the *first* random generator declared. This number is the number returned by the `make_random_generator` routine.

Return Value: The next pseudo-random number.

Note: Even a single random number generator in Canopy has several streams: in the `STREAM_PER_NODE` case, one on each node; in the `STREAM_PER_SITE` case, one for each site in addition to a separate one for the control program.

6.3.9 The Lalloc Heaps

The lalloc heaps are the memory areas for an inferior (but efficient) malloc-like function used by the `field_pointer` class routines when an off-node read occurs. There are separate heaps for the control program and for task routines, since the pointers in a task routine become invalid at the end of each site and the memory can be reclaimed, but the pointers in the control program are valid until the end of the program. Normally the default sizes and resets are adequate.

Declaring Heap Size

```
void declare_lalloc_sizes(int do_task_size,
                          int control_size);
```

Purpose: To override the default size of the lalloc heaps. This routine must be called *before* `complete_definitions`.

Arguments:

`int do_task_size`: Size in bytes of the do_task lalloc heap

`int control_size`: Size in bytes of the control lalloc heap.

Return Value: None

Resetting the Heap

```
void reset_lalloc();
```

Purpose: To reclaim all of the memory in a lalloc heap, thereby invalidating all pointers returned by `field_pointer` class routines. This resets the control lalloc heap if called from the control program or resets the do_task lalloc heap if called inside a task routine.

Arguments: None

Return Value: None

6.3.10 Quick Copy Routine

```
void intcpy(voidptr dest, voidptr src, int words);
```

Purpose: To copy data faster than the C library `memcpy` routine. This is possible for a word oriented processor, when it is known that the source and destination are word-aligned.

Arguments:

`voidptr dest`: Word-aligned pointer to destination.

`voidptr src`: Word-aligned pointer to source.

`int words`: Number of words to copy.

Return Value: None

Note: `dest` and `src` must not overlap.

6.3.11 Control of Coalescing

Vertical Coalescing

```
multithread_begin_vertical()
multithread_end_vertical()
```

Purpose: To allow a task routine to delay fetching off-node data or suspending the thread. `multithread_begin_vertical()` asserts that data from ensuing `field_pointer` calls will not be used until `multithread_end_vertical()` is called.

Warning: If these pointers are used before the `multithread_end_vertical()` call (in violation of the assertion), the results will be unpredictable. Using `move_site`, `move_site_by_path`, or `site_at_path` routines while vertical coalescing is active is illegal, and may lead to invalid pointers.

Arguments: None.

Return Value: None.

Put_field Copying Control

```
multithread_begin_nocopy()
multithread_end_nocopy()
```

Purpose: `multithread_begin_nocopy()` informs Canopy that it is safe to skip the copying of `put_field` data, because the data will remain unchanged until `multithread_end_nocopy()` is called. Allows for a smaller lalloc heap for each thread.

Warning: If the `put_field` data is changed before `multithread_end_nocopy()` is called, the data stored at the `put_field` site will be unpredictable.

Arguments: None.

Return Value: None.

6.4 User-Supplied Routines

The substance of a Canopy program will of course be the user code. This will consist of a control program (to be run on a single node) and task routines to be executed for multiple sites (and which thus may be run in parallel on multiple nodes). In addition, the user may supply routines to tailor Canopy concepts to a particular application.

For example, a typical application may use sets of sites other than entire lattices. To define the nature of these sets, the user supplies a Set of Sites function to pass to the `set_of_sites` definition routine.

Other user supplied functions include coordinate, connectivity and distribution functions used to define arbitrary "customized" grids; functions to create customized do_task keywords by defining arbitrary methods of collating data; functions defining maps from one grid to another; and functions defining customized kernels for random number generators.

Working samples of the various types of user-supplied functions can be found in the Canopy libraries. For example, GRIDLIB contains coordinate, connectivity and distribution functions used to define the prepackaged grids. These can be used as templates when an application requires custom features not available in the libraries.

6.4.1 The `control` Program

```
void control(int argc, char **argv, char **envp);
void control();
```

Purpose: The main entry point of a Canopy program. This takes the place of `main()` in a `C` program. If the application does not use arguments to the main program or environment variables, it is permissible to omit the declaration of the arguments (as per `C` conventions).

Arguments:

`int argc`: The number of command-line arguments.

`char **argv`: The array of command-line arguments, with the 0 element the name of the executing program. When a multi-node Canopy job is launched the `canopy` tool removes the number of nodes and time limit from the argument list and sets `argv[0]` to the name of the file `canopy` is executing instead of the `canopy` tool itself.

`char **envp`: An array of environment strings. This is not edited by the `canopy` tool, so the environment array is the same for a job running on the host and for a job running in a multi-node way from that host. Numerical precision may be controlled through the environment.

Return Value: None. To return a non-zero exit code call make an explicit call of `exit(code)`.

6.4.2 Do_Task Routines

All users must write *task routines* which are called on each site in some set of sites. These simple routines are the basic means of providing parallelism in Canopy programs.

More advanced users may wish to define new do_task keywords that allow more flexible communication between the control program and task routines. In Canopy all do_task keywords (except for END, which is just a number) are pointers to a CAN_do_task_keyword structure which may be defined by the user. The existing keywords are defined in chip.c and may be used as examples.

Task Routines

```
void <task_routine>(...);
```

Purpose: To do some task on a set of sites. This is the routine called by do_task.

Arguments:

... : The argument list for the task routine. These must all be pass-by-address arguments matching the call to do_task. There is a detailed description of task routines and an example in the DO_TASK section.

Return Value: None

6.4.3 Set of Sites Functions

```
int <set_of_sites_function>(grid g, intptr coords);
```

Purpose: To return the level of the site in the set or zero if the site is not in the set. For simple sets, the level is always one. This is used only by calls to `set_of_sites` and `redefine_set_of_sites` to create sets of sites. This function is called on all nodes, so if it uses any global variables (this is allowed but discouraged) they must be broadcast first.

Arguments:

`grid g`: The grid on which this set of sites is defined. Because of this argument, the set of sites function can get grid information using the grid information routines.

`intptr coords`: One-based array of coordinates describing this site

Return Value: Zero if the site is not in the set; `level` if the site is in the set. For non-compound sets, `level` is always one.

Note: The type `set_of_sites_func` is really a pointer to a function returning an `int`. In most C's, there is no distinction between pointers to functions of different types, so there is no real need to cast the function during the set-up call. However, if you want to be completely clean, the cast should be there.

Examples are on the next page:

```
Examples:
                          /* a simple set first */
int odd_func(g, coords) /* sites with odd     */
grid g;                   /* sum of coordinates */
intptr coords; {
  int i;
  int csum = 0;
  for(i=1;              /* coords is one-based */
      i<=number_of_dimensions_of_grid(g);
      i++) {
   csum += coords[i];  /* add up coordinates */
  }
  return(csum % 2);     /* 0 if even, 1 if odd */
} /* odd_func */


/* another simple set */
int even_func(g, coords)
grid g;
intptr coords; {
  return (1 - odd_func(g, coords));
} /* even_func */


/* a compound set     */
int odd_then_even_func(g, coords);
grid g;
intptr coords; {
  return (1 + even_func(g, coords));
} /* odd_then_even_func */
```

6.4.4 Lattice Definition

The `arbitrary_grid` routine uses several functions to allow the user to define a custom grid: A function (and its inverse) assigning coordinates to each site, and a connectivity function defining neighbors in each direction. In addition, the user may supply a distribution function specifying which physical node will assume responsibility for each site.

These functions are only of interest if you wish to define a new grid with new connectivity. The distribution function is not logically crucial to *defining* the grid, but a suitable choice will allow efficient processing of mostly-local applications on the grid. A default distribution function is provided, which will be adequate in most cases.

Coordinate Function

```
void <coordinate_function>(grid g,
                           int serial,
                           intptr coords);
```

Purpose: Used only at start-up time To obtain the coordinates of a site from its internal canopy `serial number`.

Arguments:

`grid g`: Grid to which the coordinates refer.

`int serial`: Internal Canopy `serial number` from 1 to the number of sites on the grid.

`intptr coords`: (Output) array of coordinates which have this internal `serial number`.

Return Value: None

Note: This had better match the inverse coordinate function.

Inverse Coordinate Function

```
void <inverse_coordinate_function>(grid g,
                                   intptr coords,
                                   intptr serial);
```

Purpose: Used both at start-up time and by `site_at_coordinates` to obtain the internal `serial` number of a site from its coordinates.

Arguments:

`grid g`: Grid where the site lives.

`intptr coords`: Coordinates of the site.

`intptr serial`: (Output) internal `serial` number of the site.

Return Value: None

Note: Examples coordinate functions and matching and inverse coordinate functions may be found in the file `grid.c`

Connectivity Function

```
void <connectivity_function>(grid g,
                             intptr coords;
                             site *site_struct;
```

Purpose: To fill the Canopy connectivity structure at set-up time and thus determine the grid's connectivity.

Arguments:

`grid g`: The grid whose connectivity is about to be calculated.

`intptr coords`: The coordinates of the site whose connectivity is about to be calculated.

`site *site_struct`: (Output.) A pointer to an array of sites which are going to be filled with nearest-neighbor information, as described in the `<canopy.h>` file. What the connectivity function does is to use the `site_at_coordinates` function (which is a part of Canopy and quite callable by ordinary users) to get nearest neighbor sites by coordinates. `Site_struct` is used as an array with elements ... `[-Y]`, `[-X]`, `[0]`, `[+X]`, `[+Y]`, ... which are filled with site variables refering to the neighbors in all directions. The 0 direction must point to itself. Other connectivities may be contructed by giving this function a different idea of the coordinates of nearest neighbors. Note that the number of directions need not be the number of dimensions and that some of the neighbors may be `NOWHERE`.

Return Value: None

An example of a connectivity function may be found in `grid.c`.

Distribution Function

```
void <distribution_function> (grid g,
                              int serial,
                              int *node,
                              intptr posit);
```

Purpose: Determines how the `arbitrary_grid` routine distributes sites to nodes. A `default_distribution_function`, provided in the GRIDLIB library

(see `grid.c`) may be used as an example to guide the creation of custom distribution functions.

Arguments:

`grid g`: The grid whose sites are to be distributed. Note that this is a hook into all of the information about grids.

`int serial`: The *serial number* of the site. Serial numbers are internal numbers Canopy uses to keep track of sites at set-up time. It is required that serial numbers be consecutive integers from 1 to the total number of sites on the grid. These serial numbers are related to site coordinates by the coordinate and inverse coordinate functions.

`int *node`: (Output) node on which to place the input site. This must be a valid node address—between 0 and `CAN_number_of_nodes`−1. See the default distribution function for correct details.

`intptr posit`: (Output) integer which is the position of the input site on the output node. `posit` must be unique for each site and must be between 1 and the number of sites on the node without any gaps. It is the responsibility of the programmer to ensure this is so!

Return Value: None

`default_distribution_function` assigns the sites in order of coordinate numbers, with the last coordinate varying most rapidly. This algorithm is recommended for most user-definied arbitrary grids. It results in reasonable efficiency for most applications on most grids. This simple strategy is easy to modify, but very hard to improve in general cases.

Using the other_params Argument

The other_params argument to arbitrary_grid (see section 6.1.1 GRID DECLARATION) may be used to pass to the user-supplied lattice definition routines additional lattice information beyond the numbers of sites, dimensions and directions and the range of each coordinate. These additional parameters are amalgamated into the array containing the lower and upper limits of the coordinates. Because the parameters are amalgamated with other information, and because the coordinates arrays are 1-based, this access is a bit tricky. The following example sets up a skew-periodic lattice– when crossing the boundary in the last dimension, the coordinates in the other dimensions shift by a skew vector. The skew vector is passed to the connectivity function via other_params. The routine uses the periodic coordinate and inverse coordinate functions appearing in grid.c, and defines its own connectivity function.

```
#include <canopy.h>
#include "../cansource/canopy_prv.h"
#include <grid.h>

void skew_periodic_connectivity_func(
  int lattice,
  intptr coords, /* 1 based!!! */
  site *site_struct)
{
  grid_list *point;  /* pointer to lattice structure */
  int i,j;
  int newcoords[MAX_COORDINATES];
  intptr skews = &point->other_parameters[-1];
  /* This is how to get at the other parameters. */
  /* We want skews[1] to correspond to others[0].*/
  point = grid_list_pointer(lattice);
  site_struct[0] = site_at_coordinates( (grid)(lattice), coords);
```

```
  for (i=0; i<point->ndir; i++) {

    /* fill +dir site_struct with +dir neighbor        */
    /* using 0-based skew periodic boundary conditions */
    for (j=0; j<point->ndir; j++) {
      newcoords[j+1]=coords[j+1];
    }
    if (coords[i+1] != point->max_coords[i]) {
      newcoords[i+1] = val + 1;
    } else {
      if (i==point->ndir-1) /* Now implement skew */
        for (j=0; j<point->ndir-1;j++) {
          newcoords[j+1] += skews[j+1];
          if (newcoords[j+1]>point->max_coords[j])
            newcoords[j+1] -= point->max_coords[j];
        }
      newcoords[i+1] = 0;
    }
    site_struct[(i+1)] =
        site_at_coordinates ( (grid)lattice, newcoords);

    /* fill -dir site_struct with -dir neighbor        */
    /* using 0-based skew periodic boundary conditions */
    for (j=0; j<point->ndir; j++) {
      newcoords[j+1]=coords[j+1];
    }
    if (coords[i+1] != 0) {
      newcoords[i+1] = val - 1;
    } else {
      if (i==point->ndir-1) /* Now implement skew */
        for (j=0; j<point->ndir-1;j++) {
          newcoords[j+1] -= skews[j+1];
          if (newcoords[j+1] < 0)
            newcoords[j+1] += point->max_coords[j];
        }
      newcoords[i+1] = point->max_coords[i];
    }
    site_struct[-(i+1)] =
        site_at_coordinates( (grid)lattice, coords);
  }
} /* periodic_connectivity_func */
```

```
grid skew_periodic_grid(int ndims,
   intptr sizes, intptr skews)
   /* sizes and skews are 1-based arrays */
{
  int i;
  int nsites;
  coordinates upper_limits;
  coordinates lower_limits;
  int others[MAXPARAMETERS];

  int n_nodes;
  n_nodes = CAN_number_of_nodes;

  for (nsites=1, i=1; i<=ndims; i++) {
    nsites *= sizes[i];
    upper_limits[i] = sizes[i]-1;
    lower_limits[i] = 0;
  }
  for (i=0; i<ndims; i++) {
    others[i] = skews[i+1];
  }
  for (i=ndims; i<MAXPARAMETERS; i++) {
    others[i] = 0; /* Set unused others to 0 */
  }
 /* call define_arbitrary_lattice to do it  */
 return(arbitrary_grid (nsites, ndims, ndims,
          lower_limits, upper_limits, others,
  default_distribution_func,
  periodic_coordinate_function,
  periodic_inv_coordinate_function,
          skew_periodic_connectivity_func) );
} /* skew_periodic_grid */
```

Users wishing to define complicated grids can also be guided by the gridlib routine `chunky_periodic_grid`, appearing in `grid.c`. Here, when the coordinate ranges and number of nodes are suitable, each node is assigned an n-dimensional "chunk" of sites, rather than the default "sheet" of sites. (This reduces the "surface to volume ratio" for each node, potentially reducing the frequency of off-node communications.)

To accomplish this, before `arbitrary_grid` is called, a routine is

called to determine the best division of sites; this produces an array of one "divisor" per coordinate. The coordinate (and inverse coordinate) function will use these divisors to create a special relation between site serial number to coordinates, implementing the improved assignments. The `other_params` mechanism is used by `chunky_periodic_grid` to make the computed divisors available to the these functions.

(The gridlib routine `chunky_periodic_grid` uses the usual periodic connectivity function and its own coordinate and inverse coordinate functions, while the `skew_periodic_grid` example uses the usual coodinate functions and its own connectivity function. Both routines use the default distribution function; clever ways of distributing sites can often be incorporated into the way the coordinate function relates serial number to coordinates.)

6.4.5 Mapping Functions

```
logical <mapfunctionname> (intptr incoords,
                           intptr outcoords);
```

Purpose: `define_map` and `read_slice_of_field` use these user-supplied routines to define maps between two grids.

Arguments:

`intptr incoords`: (Input) array of coordinates (1-based) on the domain grid of map or the field on the tape.

`intptr outcoords`: (Output) array of coordinates (1-based) on the range grid of the map or the grid supporting the field to be read. Note that there is *no information available* on the dimension or size of either the domain or range grid. The mapping function must know it internally, which means that mapping functions are really rather delicate. As a rule they must be shaped individually for each different program. However, some simple transformations, such as setting the nth coordinate to zero or returning `FALSE` for all sites with odd coordinates, may be used for many different sizes of grids.

Return Value: `FALSE` if the site at `incoords` maps to `NOWHERE` (or the site is not to be read in `read_slice`) and `TRUE` otherwise.

```
Example:   /* map sites from 4-d to 3-d */
  logical example_map(i,o)
  intptr i,o; {
     o[X] = i[Y]; /* ignore the X coord */
     o[Y] = i[Z]; /* in the original and*/
     o[Z] = i[T]; /* map all sites.     */
     return (TRUE);
  } /* example_map */
```

6.4.6 Random Number Generators

Defining Function

```
void <random_func>(random_generator_area *area);
```

Purpose: To define a random number generator. The area argument is a pointer to a structure containing the length in bytes of the state information, an `initialize` routine, and a `generator` routine. All <random_func> does is return pointers and sizes through area. Many different random number generators may be declared and the `multi_random` routine used to get numbers from different generators.

Arguments:

`random_generator_area *area`: Pointer to an area to be filled as follows:

`area->size_of_state`: size of the state area for this random number generator.

`area->generator_func`: Address of the generator function for this random number generator.

`area->initialize_func`: Address of the initialization function for the random number generator.

Return Value: None

NOTE: There are concerns about the propeties of pseudo-random sequences for use in massively parallel systems. Obviously, increased computational power means more random numbers can be used in a single stream of jobs; this argues that generators should have more bits of internal state. Less obvious is a problem that can occur if a single pseudo-random algorithm is used to generate many streams of random numbers, distinguished only by different initial states: The state of one stream can eventually match the initial state of another. For large numbers of streams, unless the period of the random sequence is extremely large, it is surprisingly likely that some pair of streams

will be related in this way. These concerns should be addressed when selecting a random number generator.

Generator Function

```
void <random_generator>(int number_to_make,
                        queue_struct *queue;
                        voidptr state;
```

Purpose: To put `number_to_make` new random numbers in `*queue`, using and updating `state`.

Arguments:

`int number_to_make`: Number of new random numbers to put in the queue. Normally, the queue is empty when this call is made.

`queue_struct *queue`: Pointer to the queue structure. This structure is defined in `canopy.c`.

`voidptr state`: Pointer to the state structure. This structure is defined privately by the random number generator, which is why it is cast as `voidptr`.

Return Value: None

Initialize Function

```
void <random_initialize>(int seed,
                         int stream,
                         voidptr state);
```

Purpose: To initialize `*state`. `complete_definitions` is the only routine that calls this.

Arguments:

`int seed`: The system-wide random number seed.

`int stream`: The stream number of this stream. Together `seed` and `stream` are used to create a unique state for this stream of the random number generator which is independent of the other streams.

`voidptr state`: The state to be filled.

Return Value: None

6.4.7 Tailoring Do_Task Keywords

Each do_task keyword such as SUM_REAL or PASS is a pointer to a CAN_do_task_keyword structure which in turn consists of pointers to various functions used by the system to implement that keyword. The user can create an instance of this structure, with poiters to user-supplied functions, customizing new do_task keywords. This section explains how that is done, and illustrates the process with an example at the end. Other examples, implementing the keywords supported by Canopy, can be found in chip.c. (The keywords are defined at the CHIP level, so that *do_task* triplets can be used by do_on_all_nodes as well as by the Canopy do_task routine.)

The elements of a do_task triplet are the *keyword* which controls the action, the *address* of the argument, and the *length* of the argument. The argument as it appears in do_task is called the *final* argument and its length the final size. The arguments of a task routine are all pointers to the argument area, called the *local* argument. The task routine is assumed to know the length (or local size) of the argument. What do_task does is to call the task routine once for each site in some set and accumulate all of the returned local arguments into the final argument, where the exact meaning of accumulate depends on the keyword. For the standard keywords, the final and local arguments have the same type and size.

Internally, do_task uses a third sort of argument called the *intermediate* argument which has the intermediate size. All of the returned local arguments are accumulated into intermediate arguments first, then the intermediate arguments on each node are accumulated together, and finally the intermediate argument on the control node is copied into the final argument. The do_task keyword points to a structure containing the functions for these operations. Note that the PASS and FUNCTION keywords work a little differently since they copy data down to the nodes instead of back up to the control program. In fact they use the same structure and routines as the accumulate arguments.

The accumulate operation must be commutative and associative, such as addition or taking the maximum value. If the computer rep-

resentation of the operation is not exactly commutative or associative then the computer will give slightly different answers depending on the number of nodes used in the calculation.

The third argument in the triplet, the final size, is present so vectors may be accumulated efficiently. A final size of zero is allowed; this does nothing. Note that local argument to the task routine will still be a valid non-null pointer in this case but that what it points to must not be written.

The most common reason for making the intermediate type different from the final type is to handle initial cases—see the `MAX_REAL` keyword definition in `chip.c`. At the end of this section is an example of a user-tailored keyword, illustrating the use of the final, local, and intermediate arguments.

In addition to functions describing how to accumulated results, the user supplies constructors, telling how to set aside room for intermediate and final results, and how to initialize them. These are conceptually similar to constructors in the C++ sense. The fact that a pointer to the created area are output as an argument rather than as the return value, and that the return value is set to the size of the allocated area, is not an important difference. An important restriction stems from Canopy calling a standardized destructor to free memory when the object constructed is no longer needed, and from assumptions made about how an intermediate or final object may be copied. Because of this, the constructor must `malloc` contiguous memory *that can be freed with a single* `free`. For instance, the intermediate constructor function can not set *inter to a pointer to a static area, or to an array of pointers to allocated blocks.

The Keyword Structure

```
typedef struct {
  int do_task_type;
  intfunptr verify_length_func;
  intfunptr inter_constructor_func;
  voidfunptr inter_accumulate_func;
  voidfunptr inter_finish_func;
  intfunptr local_constructor_func;
  voidfunptr local_accumulate_func;
} CAN_do_task_keyword, *CAN_do_task_keyword_ptr;
```

int do_task_type: This is `DO_TASK_INTEGRATE` for all of the user-defined functions. Other possible values are `DO_TASK_PASS` and `DO_TASK_FUNCTION`. `PASS` and `FUNCTION` arguments handle memory differently.

verify_length_func: Checks to see that final size is valid.

inter_constructor_func: Computes the intermediate size and `malloc`'s space.

inter_accumulate_func: Does the accumulate operation on two intermediate arguments (called only on multi-node systems).

inter_finish_func: Copies from the accumulated intermediate argument to the final argument.

local_constructor_func: Computes the local size and `mallocs`'s space.

local_accumulate_func: Accumulates a returned local value into the intermediate argument for that node.

Verify Length Function

```
logical <verify_length_function>(int bytes);
```

Purpose: Returns TRUE if `bytes` is a valid final size and FALSE if not. For keywords that allow varying lengths zero should be a valid value.

Arguments:

`int bytes`: The length in bytes passed as the third element of the `do_task` triplet.

Return Value: TRUE if the length is OK and FALSE if it is invalid.

Inter Constructor Function

```
int <inter_constructor_func>(voidptr final,
                             voidptr *inter,
                             int finalsize);
```

Purpose: Returns the intermediate size in bytes and `malloc`'s and initializes `*inter`, which is copied as an initialized empty accumulate variable to all the nodes. The memory must be allocated using a single `malloc` call.

Arguments:

`voidptr final`: Pointer to the final argument, which is the second argument of a `do_task` triplet.

`voidptr *inter`: Pointer to a pointer to the intermediate area being prepared by this function.

`int finalsize`: The size in bytes of the final argument, which is the third argument of the `do_task` triplet.

Return Value: The length in bytes of `*inter`. This is computed from `finalsize` after `finalsize` has been verified.

Inter Accumulate Function

```
void <inter_accumulate_func>(voidptr inter_acc,
                             voidptr inter,
                             int intersize);
```

Purpose: Accumulates `*inter` into `*inter_acc`.

Arguments:

`voidptr inter_acc`: Pointer to the intermediate argument which will be updated.

`voidptr inter`: Pointer to the intermediate argument with the new values.

`int intersize`: The size in bytes of intermediate arguments as returned byte the `inter_constructor_func`.

Return Value: None

Note: This is only called on multi-node systems.

Finish Function

```
void <finish_func>(voidptr final,
                   voidptr inter,
                   int intersize);
```

Purpose: Finishes the do_task operation by copying *inter to *final, editing it appropriately.

Arguments:

`voidptr final`: Pointer to the final argument. This is the second argument in a do_task triplet.

`voidptr inter`: Pointer to the intermediate argument with the completed accumulation.

`int intersize`: The intermediate size as returned by the <inter_constructor_func>.

Return Value: None

Local Constructor Function

```
int <local_constructor_func>(voidptr inter,
                             voidptr *local,
                             int intersize);
```

Purpose: Returns the local size in bytes and `malloc`'s a local area which usually requires no initialization. The memory must be allocated using a single `malloc` call.

Arguments:

`voidptr inter:` Pointer to the intermediate argument as created by `inter_constructor_func`.

`voidptr *local:` Pointer to a pointer to the local area which will be passed to the task routine. Usually this area requires no initialization but for PASS-type arguments some set-up is needed.

`int intersize:` The intermediate size as returned by the `inter_constructor_func`.

Return Value: The length in bytes of `*local`. This is computed from `intersize` after `intersize` has been computed from the original `finalsize`.

Local Accumulate Function

```
void <local_accumulate_func>(voidptr inter,
                             voidptr local,
                             int localsize);
```

Purpose: Accumulates `*local` into `*inter`.

Arguments:

`voidptr inter`: Pointer to the intermediate argument which will be updated.

`voidptr local`: Pointer to the local argument with the new values.

`int intersize`: The size in bytes of intermediate arguments as returned by the `inter_constructor_func`.

Return Value: None

Example of Customized Keyword

In this example, the user will establish a keyword MEAN_SIGMA which takes a single float value from the task routine (for each site), but returns a structure with fields representing the mean of those values and the standard deviation. Here, the lengths of the local, intermediate, and final arguments are all different: A single float goes to three intermediate quantities (a count, and sums of x and x^2), which in the finally lead to a structure containing $\overline{x}$ and σ. For purposes of illustration, and in principle to avoid loss of precision, the intermediate real results will be kept in double precision.

As is the case for any DO_TASK_INTEGRATE keyword defined, the keyword may be used in the context of a vector of results, of length of one or more elements.

Following the usual C conventions, the example will define the various user-supplied functions, followed by the keyword structure for this new integration type. The user-defined functions will be prefixed by UMS, standing for User-defined Mean and Sigma.

```
typedef struct {
  float mean;
  float sigma;
} Mean_and_Sigma;
  /* This structure holds the result:    */
  /* This typedef would be in a .h file, */
  /* to be included in the user program. */

typedef struct {
  double sum;   /* Sum of x   */
  double sumsq; /* Sum of x*x */
  int count;
} UMS_stats;
  /* This structure holds the intermediate argument */

#define FINALTYPE Mean_and_Sigma
#define INTERTYPE UMS_stats
#define LOCALTYPE float
#define FINALSIZE sizeof(FINALTYPE)
#define INTERSIZE sizeof(INTERTYPE)
```

```
#define LOCALSIZE sizeof(LOCALTYPE)

logical UMS_Word_Length(int bytes) {
  if (bytes < 0) return FALSE;
  if (bytes%4 == 0) return TRUE;
  return FALSE;
}
```

The inter_constructor_function creates and initializes an intermediate object. The final object size is used to determine how many elements there are in the vector of results. *(Note that the destructor corresponding to this constructor is implicit and assumes this function does exactly one* malloc*– the destructor does exactly one* free*).*

```
int UMS_Inter_Con(voidptr final, voidptr *inter, int finalsize) {
  int i;
  int vector_length = finalsize / FINALSIZE;
  *inter = (voidptr) malloc(vector_length*INTERSIZE);
  for (i=0; i<vector_length; i++) {  /* initialize inter object */
    ((INTERTYPE*) *inter)[i].sum   = 0.0;
    ((INTERTYPE*) *inter)[i].sumsq = 0.0;
    ((INTERTYPE*) *inter)[i].count = 0;
  }
  return(vector_length * INTERSIZE); /* return length of inter */
} /* intermediate constructor */
```

The inter_accumulate_function combines two intermediate objects. This is used both when combining results from two nodes, and within a single node if multi-thread is enabled.

```
/*inter_accumulate_function
void UMS_Accum_Stats(voidptr inter_acc, voidptr inter, int intersize)
{
  int i;
  int vector_length = intersize / INTERSIZE;
  for (i=0; i<vector_length; i++) {
   ((INTERTYPE*)inter_acc)[i].count += ((INTERTYPE*)inter)[i].n;
   ((INTERTYPE*)inter_acc)[i].sum   += ((INTERTYPE*)inter)[i].sum;
   ((INTERTYPE*)inter_acc)[i].sumsq += ((INTERTYPE*)inter)[i].sumsq;
  }
}
```

The finish_function computes the value of the final object from the value of an intermediate object. It uses intersize to determine the vector length.

```
void UMS_Find_Mean_Sigma(voidptr final, voidptr inter, int intersize)
{
  int i;
  int vector_length = intersize / INTERSIZE;
  for (i=0; i<vector_length; i++) {
    double n        = (double) ((INTERTYPE *) inter)[i].n;
    double average  = ((INTERTYPE *) inter)[i].sum/n;
    double avsquare = ((INTERTYPE *) inter)[i].sumsq/n;
    ((FINALTYPE*) final)[i].mean  = average;
    ((FINALTYPE*) final)[i].sigma =
                  sqrt((avsquare - average*average));
  }
}
```

The local_constructor_function creates a local object.

```
int UMS_Local_Con(voidptr inter, voidptr *local, int intersize)
{
  int vector_length = intersize / INTERSIZE;
  *local = (voidptr)malloc(vector_length*LOCALSIZE);
  return(vector_length * LOCALSIZE);
} /* MSDF_Local_Con */
```

The local_accumulate_function combines a local object with an intermediate object. In this case it adds to the sum and the sum of the squares of the local object, and keeps an explicit count.

```
void UMS_Accum_x_x2(voidptr inter, voidptr local, int localsize)
{
  int i;
  int vector_length = localsize / LOCALSIZE;
  INTERTYPE *interd = (INTERTYPE*) inter;
  LOCALTYPE *locald = (LOCALTYPE*) local;
  for (i=0; i<vector_length; i++) {
    double value =  ((LOCALTYPE*)local)[i];
    ((INTERTYPE*) inter)[i].n     += 1;
    ((INTERTYPE*) inter)[i].sum   += value;
    ((INTERTYPE*) inter)[i].sumsq += value*value;
  }
}
```

Finally, we can initialize the do_task keyword structure, and set up a mnemonic to refer to it:

```
CAN_do_task_keyword UMS_Mean_Sigma = {
  DO_TASK_INTEGRATE,    /* type   */
  UMS_Word_Length,      /* verify */
  UMS_Inter_Con,        /* intermediate construtor */
  UMS_Accum_Stats,      /* intermediate accumulator*/
  UMS_Find_Mean_Sigma,  /* finisher                */
  UMS_Local_Con,        /* local constructor       */
  UMS_Accum_x_x2        /* local accumulator       */
};

CAN_do_task_keyword_ptr MEAN_SIGMA = &UMS_Mean_Sigma;
```

Having defined this new type of integration, the user program could use it as follows:

```
Mean_and_Sigma r;
do_task (produce_r, mygrid,
  MEAN_SIGMA, &r, sizeof(r),
  END);
```

The produce_r task routine will have one float* argument, say xx. It after computing a result, it will set xx = result. After do_task returns, r will have r.mean set to the average and r.sigma to the standard deviation of the results returned by produce_r for each site.

6.5 CHIP Routines

CHIP, the Canopy Hardware Interface Package, is designed to isolate the Canopy implementation from machine details. It is a set of subroutines that allows machine-independent control of parallelism, and is therefore a good package for other parallel meta-applications to use. The CHIP routines and the hosting software are independent of the Canopy layer. Since Canopy is site-based not node-based, ordinary Canopy programs will have little use for these routines—but they may be included if needed.

6.5.1 Inter-Node Communication

The CHIP inter-node communication routines read and write data from and to `full_addresses`. It makes no difference if the target address is on the local node or a remote node; the operation is the same. Any piece of memory on any node can be written or read by any other node in the same job at any time, so competition must be considered in every program. The Canopy layer provides one way of organizing programs cleanly. Uses outside the Canopy paradigm must provide another way of synchronization.

The `keep`, `more`, and `close` variations of `remote_read` and `remote_write` are advisory only; they tell the communications routines that the next transfer is to the same target, so the total number of channel arbitrations may be reduced. However, `remote_read` routines always make `object` valid before returning and reads and writes are guaranteed to occur in the order specified.

Different implementations of CHIP may implement these routines in different ways. The only guarantees are these:

> Transfers are done by 32-bit words, which means that a particular 32-bit word is always some correct value. Longer entities, such as `doubles` or structures, may have invalid values since they may be part old and part new. A 32-bit word is *never* set to any other value than the old value or the new value (such as by being cleared to 0 first). Flags should therefore always be 32-bit words.

> Any memory access after a `remote_write`, local or remote, will read the changed value. Any memory access after any local write (even one not using the `remote_write` routines) will read the changed value. This means that—unlike some implementations of NFS—it will always work in Canopy to change a value, tell other nodes the value is changed, and know that if they read it they will read the new value (or some later value). This is true even of the `keep` variations.

> While transfers (except for single-word transfers) need not be atomic, the word with the highest address will always be changed after all the other words. indextransfers!non-atomic Even if the keep routines are used, the highest word of the previous transfer will be done before any word of the next transfer. Of course, it is preferable to write a data block and then write a flag in a separate transfer since this is easier to modify correctly.

Transfers may take arbitrary lengths of time. One node may access another several times, or none, before a third node can. The order of transfers except for the last word is undefined.

Operations that need exclusive access should use semaphores. Here is an example used in setting up maps in Canopy where nodes need to update a counter asynchronously. This code works regardless of whether where is local or remote.

```
                /* start an atomic operation */
wait_for_resource(&where_semaphore);
          /* keep and close are advisory only */
remote_read_and_keep(where, 1, &what);
what += 1;
remote_write_and_close(where, 1, &what);
               /* let somebody else update it */
free_resource(&where_semaphore);
```

Single-block Communication

```
void remote_read
void remote_read_and_keep
void remote_read_more
void remote_read_and_close
       (full_address *add, int len, voidptr object);

void remote_write
void remote_write_and_keep
void remote_write_more
void remote_write_and_close
       (full_address *add, int len, voidptr object);
```

Purpose: To read `len` words from `*add` into `object` or to write `len` words to `*add` from `object`. The `keep`, `close`, and `more` variations give hints to the system about whether more data is going to the same node.

Arguments:

`full_address *add:` The source (read) or destination (write) address. The node may be any node including the calling node but the address must be word-aligned.

`int len:` The number of words to move.

`voidptr object:` A local pointer to the destination (read) or source (write) of the data on this node. This must be word-aligned.

Return Value: None

NOTE: Do not rely on the `keep` variations to act as semaphores. Use the `semaphore` routines if semaphores are needed.

Multiple-block Communication

```
int remote_gather  (int nblocks, int* node,
                    voidptr *src, int *len, vodiptr *dst)
```

Purpose: To read `nblocks` blocks of data of various lengths from a list of addresses (`src`) on a specified remote node, into memory blocks (`dst`) on the local node. The remote gather and scatter routines facilitate transfer coalescing, in which several transfers between two nodes are done in one transaction on the communications medium. Although remote_gather and remote_scatter could be written in terms of multiple calls to remote_read and remote_write, the point of these routines is to reduce the number of communications.

To further facilitate efficient communications in applications which may involve high levels of traffic, this transfer is non-blocking: if the channel is busy, control may return immediately, with a negative return value.

Arguments:

`int nblocks`: The number of blocks of data to be transfered.

`int *node`: The remote node to gather data from. (May be identical to the local node, in which case simple memory copies are done.)

`voidptr *src`: A local pointer to a list of source addresses. Each is to be used as an address of a block of data on the remote node.

`int *len`: A local pointer to a list of lengths (numbers of words) of each block to transfer.

`voidptr *dst`: A local pointer to a list of destination addresses. Each is to be used as an address on the local node for a block of data to be copied.

Note: Transfers must be word aligned: The elements of the source and destination lists must be multiples of four.

Return value:: Either a negative integer error code, or the actual number of blocks read. The only circumstance in which an error can be returned is if the connection is refused (because the channel is busy) and should be re-tried later; all other errors result in an abort of the job. The actual number of blocks read can be less than nblocks, if nblocks is greater than `CAN_ATOMIC_GATHER` (defined in `chip.h`).

```
int remote_scatter (int nblocks, int* node,
                    voidptr *dst, int *len, vodiptr *src)
```

Purpose: To write `nblocks` blocks of data of various lengths from a list of addresses (`src`) on the local node, into memory blocks (`dst`) on the specified remote node.

Arguments:

`int nblocks`: The number of blocks of data to be transfered.

`int *node`: The remote node to gather data from. (May be identical to the local node, in which case simple memory copies are done.)

`voidptr *dst`: A local pointer to a list of destination addresses. Each is to be used as an address on the remote node for a block of data to be copied.

`int *len`: A local pointer to a list of lengths (numbers of words) of each block to transfer.

`voidptr *src`: A local pointer to a list of source addresses. Each is to be used as an address of a block of data on the local node.

Note: Transfers must be word aligned: The elements of the source and destination lists must be multiples of four.

Return value:: Either a negative integer error code (if the channel was busy), or the actual number of blocks written.

6.5.2 Full Address Functions

```
void full_address_from_local_address(voidptr local_address);
```

Purpose: To form a `full address` structure pointing to a location on the local node specified by a local pointer.

Arguments:

`voidptr local_address`: The address in local memory space.

Return Value: A structure of type `full_address` pointing to the local address on this node.

```
void full_address_on_another_node
                    (voidptr address, int node_number);
```

Purpose: To form a `full_address` structure pointing to a location on an arbitrary node, specified by the node number and an address within that node's memory space.

Arguments:

`voidptr address`: The address in in node memory space; an ordinary pointer when used by that node.

`int node_number`: Specifies the node associated with the memory of the desired location.

Return Value: A structure of type `full_address` pointing to the specified address on the specified node.

6.5.3 Fatal Errors and Memory Allocation

```
void yderr(char *errstring);
```

Purpose: To report fatal errors. This routine stops the program when called from any level and reports `*errstring` to the controller.

Arguments:

`char *errstring`: The error message to report to the controller.

Return Value: None. This function doesn't even return!

```
voidptr ckmalloc(int size);
```

Purpose: To allocate memory without requiring the user to check for a null return. This routine behaves identically to `malloc()` excpet that a yderr() is declared if insufficient memory is available.

Arguments:

`int size`: The number of bytes to be allocated.

Return Value: A valid pointer to the allocated block of memory. If memory could not be allocated, this function will call `yderr()` with a suitable error string, and never return.

6.5.4 Do On All Nodes

```
void do_on_all_nodes(voidfunptr local_func,
                     <triplets>,
                     END);
```

Purpose: To do `local_func` on all nodes of the system.

Arguments:

`voidfunptr local_func:` A pointer to a function whose arguments aare those defined by the `<triplets>` given. This function will be called once on each node. The arguments work the same way as for `do_task` (see section 6.2.1 DO_TASK TRIPLETS).

`triplets:` These are the same as the `do_task` triplets described with `do_task`.

`END:` This is the `END` keyword used just the same was as in `do_task`.

Note: This function works at the CHIP level, not at the Canopy level, which means that Canopy concepts such as the `HOME` site and `field_pointer` routines are not valid in `local_func`. The random number generator will not work there either. To use Canopy constructs on a per-node basis define a `grid` containing as many sites as there are nodes and then do a `do_task` over that grid.

Note2: There is an CHIP function called `do_frame_on_all_nodes` which is to this function as `vprintf` is to `printf`, used for the implementation of the various `do_task` routines. This private function uses a `frame` argument which is non-trivial to set up and subject to change. However, it must be used to build a system such as Canopy where `vprintf`-like functionality is needed.

Return Value: None

6.5.5 Semaphores and Resources

```
logical init_resource(full_address *resource);
logical lock_resource(full_address *resource);
void wait_for_resource(full_address *resource);
void free_resource(full_address *resource);
```

Purpose: To provide a CHIP standard way of contending for resources. In CHIP, each resource is assigned a `semaphore` variable which governs it. The full-addresses used as arguments to these routines must point to that semaphore variable. Before the semaphore can be used, it must be initialized with a call to `init_resource`. After that call the `lock`, `wait_for`, and `free` routines may be used to control the resource. If the `semaphore` variable was automatic or `malloc`'ed it must not be used after it goes out of scope (meaning that the routine automatically allocating it returns—it is OK to use it in that routine's subroutines, including tasks that routine spawns) or is `free`d.

Arguments:

`full_address *resource`: Address of a full-address pointer to a semaphore variable.

Return Value: TRUE if the request was granted and FALSE if the semaphore was busy or could not be initialized.

Note: The CHIP semaphore contention is not guaranteed to be fair, especially if `wait_for_resource` is used. It will always be mutually exclusive and will never deadlock.

Chapter 7

Canopy Libraries

The Canopy libraries provide routines for operations which are ubiquitous over a wide range of applications. These include functionallity particular to Canopy (e.g. `gridlib`), generic to massively parallel systems (e.g. `ranlib`), or needed because of peculiarities in I/O or deficiencies in the C language (e.g. `cmplxlib`). The utility of these routines extends beyond the obvious advantage of not having to re-create each routine:

- When common routines are standardized, there is an advantage in code readability and in confidence in routine correctness.
- Any subtle issues which arise in implementing these routines can be resolved correctly once and for all.
- In cases where efficiency is likely to be an issue, the library routines might be more carefully optimized than individually created routines would be.

Other libraries of routines oriented toward more specific applications groups are also available. An example is QCDLIB, which contains routines for manipulation of SU(3) and GL(3) matrices and 3-vectors. These applications libraries are not part of Canopy *per se*, and therefore are not documented in this manual.

7.1 Gridlib—Periodic Grids

The grid library contains standard periodic grid defining functions as described in the grid declaration section of CANOPY SUBROUTINE REFERENCE. The sources for these routines are also useful as templates for customized grids.

The grid library `#defines` names for the directions in `grid.h`, which should be `#included` in any program using these routines:

```
#define X 1
#define Y 2
#define Z 3
#define T 4
#define MINUS_X -1
#define MINUS_Y -2
#define MINUS_Z -3
#define MINUS_T -4
```

7.1.1 Periodic Grids

```
grid periodic_linear_grid(int x);
grid periodic_square_grid(int x, y);
grid periodic_cubic_grid(int x, y, z);
grid periodic_hypercubic_grid(int x, y, z, t);
```

indexgrids!periodic linear indexgrids!periodic square indexgrids!periodic cubic indexgrids!periodic hypercubic

Purpose: To declare rectilinear periodic grids with coordinates in the range `[0..n-1]`. These functions map the sites to the serial numbers (and hence to the nodes) with x varying fastest, then y, then z, and finally t. The result is that sites in the same time slice tend to be together on the nodes.

Arguments:

`int x, y, z, t:` The x, y, z, and t dimensions of the new periodic grid. The grid coordinates run `[0..x-1]` `[0..y-1]` ....

Return Value: The `grid` variable referring to this grid, which is used by all the subroutines using this grid.

7.1.2 Chunky Periodic Grids

```
grid chunky_periodic_square_grid(int x, y);
grid chunky_periodic_cubic_grid(int x, y, z);
grid chunky_periodic_hypercubic_grid(int x, y, z, t);
```

Purpose: To declare rectilinear periodic grids with coordinates in the range `[0..n-1]`. These functions map the sites to the serial numbers (and hence to the nodes) in a way that tries to keep neighboring sites on the same nodes. Each dimension except the last is divided into even chunks (so the size shouldn't be prime) and then the chunks are divided by the distribution function. The net result is that approximately a square chunk of the grid is on each node, which maximizes the likelihood that nearest neighbors will be on the same node.

Arguments:

`int x, y, z, t:` The x, y, z, and t dimensions of the new periodic grid. The grid coordinates run `[0..x-1] [0..y-1] ....`

Return Value: The `grid` variable referring to this grid, which is used by all the subroutines using this grid.

7.1.3 General Periodic Grids

```
grid periodic_grid(int ndims, intptr size);
grid chunky_periodic_grid(int ndims, intptr size);
```

Purpose: To declare rectilinear grids of arbitrary dimensions with zero-based coordinates. To declare a grid whose lowest coordinate is not 0 use the `arbitrary_grid` routine.

Arguments:

`int ndims`: The number of dimensions of the new grid.

`intptr size`: A one-based array of sizes. The coordinates of the new grid run `[0..size[1]-1]`, `[0..size[2]-1]`, ... `[0..size[ndims]-1]`. This agrees with the normal definitions of directions so that `size[X]` is the first direction and so forth for `Y`, `Z`, and `T`. Note that if `size` was declared as `coordinates` that `size[0]` exists but is not used. If `size` was an array returned from some function call (such as `grid_lower_bounds`) then `size[0]` may not exist.

Return Value: The `grid` variable referring to this grid.

7.2 Setlib—Predefined sets

Several set-of-sites functions, such as `red_func` which selects those sites whose sum of coordinates is odd, are quite common. They have been written in a clean way and collected here so users don't have to keep re-writing them. An attempt has been made to keep them general enough so they will work on any grid. The `#include` file is `set.h` and the C switch is `-lset`.

All of these functions have the standard syntax of a set-of-sites function:

```
int <set_of_sites_func>(grid lattice, intptr coords);
```

where `lattice` is the grid on which the set is defined and `coords` is the (1-based) array of coordinates for a particular site. The return value is either the level of the site in the set or 0 if the site is not in the set. The following functions are provided:

`red_func` : The set of sites with odd sum of coordinates.

`black_func` : The set of sites with odd sum of coordinates.

`rb_func` : The compound set of sites red first then black.

`br_func` : The compound set of sites black first then red.

`hyperl_func` : The compound set of sites to make a hyperplane. In this set the level of each site is the sum of its coordinates plus one.

`hyperu_func` : The compound set which is `hyperl_func` in the reverse order.

`sphere1_func` : The compound set in which the level of each set is its distance from the origin plus one.

`sphereu_func` : The compound set which is `sphere1_func` in the reverse order.

7.3 Ranlib—Random Numbers

The random number library contains the functions needed to set up a Canopy random number generator. The `#include` file is `random.h` and the C switch is `-lran`. Issues involving random numbers in a multiprocessing context are discussed in section 4.7.1 RANDOM NUMBERS. There are currently two random number generator functions in the library:

`bad_random()`: Created in the dark ages as a test, `bad_random` uses a modular congruence algorithm with a short period.

`dual_random()`: Created 5/25/88 by Doug Toussaint. This is a feedback shift register generator xor'ed with a modular congruence

The way to use either of these in a Canopy program with a single random number generator is (*cf.* section 6.1.5 RANDOM NUMBER DECLARATION and section 6.3.8 OBTAINING RANDOM NUMBERS):

```
#include <random.h>
...
make_random_generator(dual_random,STREAM_PER_SITE,<n>,<seed>);
...
complete_definitions();
```

Having prepared this, then calls to `random()` (by either task routines or the control program) will return floating point numbers between 0.0 (inclusive) and 1.0 (exclusive). In place of `dual_random`, `bad_random`, (or a user-defined generator) could be selected; and `STREAM_PER_NODE` randoms may be chosen instead of `STREAM_PER_SITE`.

For efficiency, generators produce several randoms at a time and dole them out each time `random()` is called; `<n>` controls the number of randoms to generate at once (10 is good typically). `<seed>` is an integer seed which labels a stream of random numbers.

7.4 Cmplxlib—Complex Arithmetic

The cmplx library handles complex numbers, both in single and double precision. The line `#include <cmplx.h>` must be in any program using these routines, and it automatically includes the `canopy.h` file. Therefore the `#include <cmplx.h>` line must be the first include file in any program using these functions. Use the `-lcmplx` switch to include this library if you are not using the Canopy shells which include it automatically.

7.4.1 Complex Numbers

There is no intrinsic `complex` type in C, so one is defined here, along with the normal functions on the complex domain.

```
typedef struct { /* single precision */
   float real;
   float imag;
} complex;

typedef struct { /* double precision */
   double real;
   double imag;
} double_complex;
```

Because complex operations are frequently in time-critical places, both functions and macros have been provided to handle complex operations. Due to some compiler's difficulties in handling nested functions returning structures, all of the functions take as their input a pointer to a complex number. The arguments for the macros are complex numbers, not pointers, but like all macros they evaluate their arguments many times.

7.4.2 Complex Functions

```
complex cmplx(float r, float i);
complex cadd(complex *a, complex *b);
complex cmul(complex *a, complex *b);
complex csub(complex *a, complex *b);
complex cdiv(complex *a, complex *b);
complex conjg(complex *a);
float cabs(complex *a);
float cabs_sq(complex *a);
float carg(complex *a);
```

Purpose: To initialize, add, multiply, subtract, divide, conjugate, find the magnitude and find the phase of complex numbers.

Arguments:

`float r`: Real part of result.

`float i`: Imaginary part of result.

`complex *a`: Pointer to the first complex argument.

`complex *b`: Pointer to the second complex argument.

Return Values: The result of the operation. `cabs_sq` returns the square of the magnitude of `*a`; `carg` returns the phase of `*a`.

```
complex csin(complex *z);
complex ccos(complex *z);
complex ctan(complex *z);
complex csinh(complex *z);
complex ccosh(complex *z);
complex ctanh(complex *z);
complex casin(complex *z);
complex cacos(complex *z);
complex catan(complex *z);
complex casinh(complex *z);
complex cacosh(complex *z);
complex catanh(complex *z);
complex cexp(complex *a);
complex clog(complex *a);
complex csqrt(complex *a);
complex ce_itheta(float theta);
```

Purpose: Complex transcendental functions.

Arguments:

`complex *z`: Pointer to the complex argument.

`float theta`: Angle.

Return Values: The complex result of the operation.

`csin, ccos, ctan`: trig functions.

`csinh, ccosh, ctanh`: hyperbolic functions.

`casin, cacos, catan`: inverse trig functions.

`casinh, cacosh, catanh`: inverse hyperbolic functions.

`cexp, clog, csqrt`: exponentional, natural logarithm, and square root functions.

`ce_itheta`: Returns the unit complex number with angle theta.

7.4.3 Double Complex Functions

```
double_complex dcmplx(double r, double i);
double_complex dcadd(double_complex *a, *b);
double_complex dcmul(double_complex *a, *b);
double_complex dcsub(double_complex *a, *b);
double_complex dcdiv(double_complex *a, *b);
double_complex dconjg(double_complex *a);
double dcabs(double_complex *a);
double dcabs_sq(double_complex *a);
double dcarg(double_complex *a);
```

Purpose: To initialize, add, multiply, subtract, divide, conjugate, find the magnitude and find the phase of double-precision complex numbers.

Arguments:

`float r`: Real part of result.

`float i`: Imaginary part of result.

`double_complex *a`: Pointer to the first argument.

`double_complex *b`: Pointer to the second argument.

Return Values: The result of the operation. `cabs_sq` returns the square of the magnitude of `*a`; `carg` returns the phase of `*a`.

```
double_complex dcsin(double_complex *z);
double_complex dccos(double_complex *z);
double_complex dctan(double_complex *z);
double_complex dcsinh(double_complex *z);
double_complex dccosh(double_complex *z);
double_complex dctanh(double_complex *z);
double_complex dcasin(double_complex *z);
double_complex dcacos(double_complex *z);
double_complex dcatan(double_complex *z);
double_complex dcasinh(double_complex *z);
double_complex dcacosh(double_complex *z);
double_complex dcatanh(double_complex *z);
double_complex dcexp(double_complex *a);
double_complex dclog(double_complex *a);
double_complex dcsqrt(double_complex *a);
double_complex dce_itheta(float theta);
```

Purpose: Double-precision complex transcendental functions.

Arguments:

`double_complex *z`: Pointer to the argument.

`float theta`: Angle.

Return Values: The double-precision complex result of the operation.

`dcsin, dccos, dctan`: trig functions.

`dcsinh, dccosh, dctanh`: hyperbolic functions.

`dcasin, dcacos, dcatan`: inverse trig functions.

`dcasinh, dcacosh, dcatanh`: inverse hyperbolic functions.

`dcexp, dclog, dcsqrt`: exponentional, natural logarithm, and square root functions.

`ce_itheta`: unit complex number with angle theta.

7.4.4 Complex Macros

```
CONJG(a,b)          b = conjg(a)
CADD(a,b,c)         c = a + b
CSUM(a,b)           a += b
CSUB(a,b,c)         c = a - b
CMUL(a,b,c)         c = a * b
CDIV(a,b,c)         c = a / b
CMUL_J(a,b,c)       c = a * conjg(b)
CMULJ_(a,b,c)       c = conjg(a)*b
CMULJJ(a,b,c)       c = conjg(a*b)
CNEGATE(a,b)        b = -a
CMUL_I(a,b)         b = ia
CMUL_MINUS_I(a,b)   b = -ia
CMULREAL(a,f,c)     c = fa
CDIVREAL(a,f,c)     c = a/f
```

Purpose: In-line Macros for fast complex operations.

Arguments:

`a, b, c`: `complex` or `double_complex` numbers—not pointers!

`f`: A `float` or `double` for the real operations.

Return Values: None. These macros return nothing.

Note: These are C macros, which means that there are restrictions on the arguments:

1: Since the macro arguments may be evaluated more than once they must not be functions with side effects.

2: For `CMUL`, `CDIV`, `CMUL_J`, `CMULJ_`, `CMULJJ`, `CMUL_I` and `CMUL_MINUS_I` the result must not be one of the other two arguments. Any of the other arguments to any of the macros may be the same.

7.4.5 Polynomial Evaluation

```
complex evaluate_complex_polynomial
   (int order, complex *coef, complex *val);
double_complex evaluate_double_complex_polynomial
            (int order, double_complex *coef, *val);
```

Purpose: To evaluate a polynomial.

Arguments:

`int order`: The order of the polynomial (such as 3 for cubic).

`(type) *coef`: An array of `order` + 1 numbers which are the coefficients of the polynomial : `coef[order]`x^{order} + ... + `coef[0]` = 0.

`(type) *val`: A pointer to the number to be plugged into the polynomial.

Return Value: The result of the evaluation.

7.4.6 Root Polishing

```
complex polish_complex_root
    (int order, complex *coef, complex *root, int n);
double_complex polish_double_complex_root
     (int order, double_complex *coef, *root, int n);
```

Purpose: To improve a putative polynomial root by the Newton-Raphson method. Note that the putative roots are always in the complex domain.

Arguments:

`int order`: The order of the polynomial (such as 3 for cubic).

`(type) *coef`: An array of `order` + 1 numbers which are the coefficients of the polynomial : `coef[order]`x^{order} + ... + `coef[0]` = 0.

`(type) *root`: A pointer to the putative root which will be improved.

`int n`: The number of times to iterate the Newton-Raphson procedure.

Return Value: The improved root. Note that this routine comes with no guarantees—if the putative root is not close to an actual root or if the polynomial is not stable then the returned value may not be closer to a root. A good value of n is about 10.

7.4.7 Quadratic Equation

```
int solve_complex_quadratic(complex *coef, *root);
int solve_double_complex_quadratic
      (double_complex *coef, *root);
```

Purpose: To solve the quadratic equation.

Arguments:

(type) *coef: An array of three numbers which are the coefficients of the quadratic : coef[2]x^2 + coef[1]x + coef[0] = 0.

(type) *root: An array of two complex numbers to hold the returned roots of the equation (starting at root[0]).

Return Value: The magnitude of the return value is the number of non-degenerate roots. If *both* roots are real the return value is negative, otherwise it is positive.

7.4.8 Cubic Equation

```
int solve_complex_cubic(complex *coef, *root);
int solve_double_complex_cubic
      (double_complex *coef, *root);
```

Purpose: To solve the cubic equation.

Arguments:

(type) *coef: An array of four numbers which are the coefficients of the cubic equation:
coef[3]x^3 + coef[2]x^2 + coef[1]x + coef[0] = 0.

(type) *root: An array of three complex numbers to hold the returned roots of the equation (starting at root[0]).

Return Value: The magnitude of the return value is the number of non-degenerate roots. If *all* roots are real the return value is negative, otherwise it is positive. If two roots are degenerate, they are root[1] and root[2].

7.5 FFTlib—Fast Fourier Transforms

The FFT library provides subroutines which do Fourier transforms and inverse Fourier transforms on rectangular periodic grids of any size and dimension less than seven. The sizes in each direction are *not* limited to a power of two, but the current implementation is extremely slow unless the size is 2, 3, or 5 times a power of two.

Since for certain analysis programs, it is useful to FFT just the spatial directions, this library also includes routines to transform only specified directions.

To use this library, include `<fft.h>` in your program and then call `FFT_setup` once for each `grid` on which you wish to do Fourier transforms. After `complete_definitions`, call `FFT` or `Inverse_FFT` with the `field` you wish to transform, which must live on one of the `grids` you declared. The field length must be a multiple of `sizeof(complex)`. Canopy will treat the field as a vector of complex numbers and transform each element of the vector. The order of the transformed field is the natural one, with element (x,y,z,t) corresponding to the element with momentum (x,y,z,t) in units of 2π/(number of elements in that direction).

Here is a sample program using this library:

```
#include <fft.h>
grid g1, g2; field f1, f2, f3;
void control() {
g1 = periodic_cubic_grid(4,4,8);
g2 = periodic_hypercubic_grid(8,8,8,12);
f1 = site_field(g1,2*sizeof(complex));
f2 = site_field(g2,1*sizeof(complex));
f3 = site_field(g1,3*sizeof(complex));
FFT_setup(g1); FFT_setup(g2);
...
complete_definitions();
... /* initialize f1, f2, and f3 */
FFT(f1);
Inverse_FFT(f2);
FFT(f3);
Inverse_FFT(f3);
FFT(f2);
Inverse_FFT(f1); /* back the way it was */
}
```

On the first call to either FFT or Inverse_FFT for each grid an initialization subroutine is done which more than doubles the time of the first transform. Subsequent calls are therefore faster.

```
void FFT_setup(grid g);
void FFT(field f);
void FFT_double(field f);
void Inverse_FFT(field f);
void Inverse_FFT_double(field f);
```

Purpose: To do Fourier transforms on fields. The basic routines treat the field elements as collections of floats; the "double" routines treat them as collections of doubles.

Arguments:

`grid g`: For the setup, the grid on which the fields to be transformed live. One call must be make for each grid on which transforms are to be done. This routine must be called before `complete_definitions`.

`field f`: The field to transform. This field must live on one of the grids with which `FFT_setup` was called.

Return Values: none.

```
void FFT_some_directions
        (field f, logical *fft_this_dir);
void FFT_double_some_directions
        (field f, logical *fft_this_dir);
void Inverse_FFT_some_directions
        (field f, logical *fft_this_dir);
void Inverse_FFT_double_some_directions
        (field f, logical *fft_this_dir);
```

Purpose: To do Fourier transforms on some directions on fields. These routines operate the same way as ordinary FFT routines, excpet that the transform is only performed for direction `d` if `fft_this_dir[d]` is `TRUE`.

Arguments:

`field f`: The field to transform. This field must live on one of the grids with which `FFT_setup` was called.

`logical *fft_this_dir`: An array specifying which directions are to be transformed.

Return Values: none.

7.6 Promptlib—Extended Input

These routines provide standard robust user-friendly flexible ways to read `stdin` into Canopy programs.

- They ensure that the expected number of items is read.
- They can range check the input arguments.
- They can provide default values for input arguments.
- They recover from errors in the input.
- They echo the input if the input is from a file, which makes the output look much better.
- They ask the user to re-enter the input if an error is made at a terminal, but they exit if an error is made in input from a file.

For Canopy applications running on distributed systems, the promptlib routines have the further advantage of resolving some sticky problems with ordinary I/O, for instance, what happens if an interrupt is encountered.

The `#include` file is `prompt.h` and the C switch is `-lprompt`.

7.6.1 Example Using Prompts

The file doprompt.c in cansource/programs contains examples illustrating how prompted input may be used. It is self-explanatory:

```
#include <canopy.h>
#include <prompt.h>
#include <math.h>

void control()          /* example of the use of prompts */
{
  float vals[10];   /* The first part reads in up to 10 */
  int d[5];         /* floats and then prompts for a    */
  int i,j,k;        /* power for each one.              */
  complete_definitions(); /* this is in all Canopy progs*/

                    /* Get the list of up to 10 numbers */
  i=prompt_float_list("Enter up to 10 numbers: ",10,vals);
  for (j=0; j<i; j++) { /* get the power for each one   */
    char ss[80];        /*with a default value and range*/
    float power;
    sprintf(ss,"Power for %f ",vals[j]);  /* make prompt*/
    power = prompt_float_range_default(ss,1.0,0.1,100.0);
    printf("%2d: %g^%f is %g\n",j,vals[j],
                                power,pow(vals[j],power));
  }
                    /* This is how to read in exactly n */
                    /* values in a comma-delimited list */
  prompt_scanf(4, "Enter four integers: ",
               "%d,%d,%d,%d",&d[1],&d[2],&d[3],&d[4]);
  printf("The 4-d point is (%d,%d,%d,%d)\n",
                                  d[1],d[2],d[3],d[4]);
}  /* control */
```

To compile and link this example for a Unix system and for the D860 nodes in ACPMAPS, use

```
canc doprompt.c -o doprompt
dcanc doprompt.c -o doprompt
```

Note that -lprompt is included automatically by the shells.

A sample session using this file from a terminal is:

```
>doprompt
Enter up to 10 numbers: 1.1,2,4.3e5
Power for 1.100000 [0.1..(default 1)..100] 6
 0: 1.1^6.000000 is 1.77156
Power for 2.000000 [0.1..(default 1)..100] 0.07
Value out of range -- re-enter
Power for 2.000000 [0.1..(default 1)..100] 2
 1: 2^2.000000 is 4
Power for 430000.000000 [0.1..(default 1)..100]
 2: 430000^1.000000 is 430000
Enter four integers: 1,2,3,4
The 4-d point is (1,2,3,4)
>
```

Notice how the out-of-range input "0.07" was re-prompted and how the default value "1" was taken.

7.6.2 Subroutines in the Prompt Library

As outlined in the introduction, the prompt library provides clean input subroutines. All of these routines prompt `stdout` with the prompt string (which may be the null string) and fetch one line of input from `stdin`. This line is then checked to ensure it contains the requested input tokens and no others. If it does then the value is either returned or put in an argument as described in the individual routines. If it does not, the routine takes different action depending on whether `stdin` is a terminal device or not. For terminals, the routines do the following: print "`Input is in error -- please re-enter`" on `stdout`; print the prompt string again; and then wait for a new line to be typed. If `stdin` is a file, they print "`Invalid input in file -- quitting`" and exit the program.

If only a newline is entered, the routines which have default values will return the default value. If the input is out of range on those routines that check range the error is handled like any other invalid entry.

These routines place a length limitation on the input line of 255 characters.

All of these routines, including the ones that return lists of values, make sure there is no extra garbage at the end of the line. They also echo the input line if `stdin` is a file and do not echo it if `stdin` is a terminal. This makes `stdout` look the same with both types of input since the terminal handler echos the input line.

Prompt for Single Value

```
double prompt_double(char *prompt);
float prompt_float(char *prompt);
int prompt_int(char *prompt);
void prompt_string(char *prompt, char *ss);
void prompt_word(char *prompt, char *ss);
```

Purpose: To prompt for and read in a single value. The differencce between `prompt_word` and `prompt_string` is that `prompt_word` expects a single word with no white space characters inside it and `prompt_string` returns the entire input line. Note that neither string routine will return a null string—if a return is entered the user will be prompted again (if on a terminal) or the program will exit (if input is from a file). Promptlib has other routines that interpret a null string (entered by just pressing return) as taking the default value—these are described later.

Arguments:

`char *prompt`: The prompt string. This is used verbatim.

`char *ss`: The address of memory for the returned string.

Return Value: For `prompt_int`, `prompt_float`, and `prompt_double` the return value is the user input value.

Prompt with Default Value

```
double prompt_double_default(char *prompt,
                             double default);
float prompt_float_default(char *prompt,
                           double default);
int prompt_int_default(char *prompt,
                       int default);
void prompt_string_default(char *prompt,
                           char *default,
                           char *ss);
void prompt_word_default(char *prompt,
                         char *default,
                         char *ss);
```

Purpose: To prompt for and read in a single value, taking the default value if a newline is entered.

Arguments:

`char *prompt`: The prompt string. The string

```
"(default <default>) "
```

is appended to the prompt.

... `default`: The default value. Note that this is a `double` for the float routine; this is because some versions of C don't always handle float arguments correctly. With this declaration, no cast is needed on the calling line.

`char *ss`: The address of memory for the returned string.

Return Value: For `prompt_int_default`, `prompt_float_default`, and `prompt_double_default` the return value is the user input value.

Prompt with Range Check

```
double prompt_double_range(char *prompt,
                           double low,
                           double high);
float prompt_float_range(char *prompt,
                         double low,
                         double high);
int prompt_int_range(char *prompt,
                     int low,
                     int high);
```

Purpose: To prompt for and read in a single value checking that it is in a given range. If the input is not in the range, it is prompted for again (if terminal input) or the program is terminated (if file input).

Arguments:

`char *prompt`: The prompt string. The string
`"[<low>..<high>] "`
is appended to the prompt.

... `low`: The lower limit.

... `high`: The upper limit.

Note: The range accepted is `low` $\leq$ value $\leq$ `high`.

Return Value: The user input value.

Prompt with Default and Range Check

```
float prompt_double_range_default(char *prompt,
                                  double default,
                                  double low,
                                  double high);
float prompt_float_range_default(char *prompt,
                                 double default,
                                 double low,
                                 double high);
int prompt_int_range_default(char *prompt,
                             int default,
                             int low,
                             int high);
```

Purpose: To prompt for and read in a single value checking that it is in a given range and giving the default value if a newline is entered.

Arguments:

`char *prompt`: The prompt string. The string

```
"[<low>..(default <default>)..<high>] "
```

is appended to the prompt.

... `default`: The default value.

... `low`: The lower limit.

... `high`: The upper limit.

Note: The range accepted is `low` $\leq$ value $\leq$ `high`.

Return Value: The user input value.

Prompt for a List of Values

```
int prompt_double_list(char *prompt,
                       int max,
                       double* list);
int prompt_float_list(char *prompt,
                      int max,
                      float* list);
int prompt_int_list(char *prompt,
                    int max,
                    int* list);
```

Purpose: This prompts for and reads in some number of comma-separated items. It returns the number of items in the list. One use of this in lattice gauge theory is to input the size of the lattice. The maximum number of items that can be read in is the maximum number that can fit on one line.

Arguments:

`char *prompt`: The prompt string. This is used verbatim.

`int max`: The maximum number of items that may be in the list. If more are entered, the routine prompts again with a message of what the maximum is. In the current implementation the maximum value for `max` is 20.

`xxx *list`: A 0-based array of whatever type was specified.

Return Value: The number of items entered.

General Prompt

```
void prompt_scanf(int expected,
                  char *prompt,
                  char *format,
                  ...)
```

Purpose: This is the most general promptlib routine. It expects some number of arguments and keeps re-prompting until it gets them (unless the input is from a file, in which case it exits the program with an error message). The format and variable arguments are identical to those used in `scanf` except that the format may not contain any newline characters.

Arguments:

`int expected`: The expected number of return values. Unfortunately this may not match the number of items in the format string or the argument list, so count carefully or use one of the other routines.

`char *prompt`: The prompt string. This is used verbatim.

`char *format`: The format string. This is the same as the format string for `scanf` except that it must contain no newline characters. It should have as many fields as the expected number of arguments.

`...` The addresses of the values being read, as in `scanf`.

Return Value: There is no return value from this function since it keeps retrying until it succeeds (if an error occurs in terminal input) or it stops the program (if an error occurs in file input).

Index

Canopy Version 7.0 Quick Reference Section

CANOPY Types:

void	void *voidptr
int *intptr	logical
float *floatptr	char *charptr
grid	field
set	map
site	path
direction	coordinates
full_address	field_address
sync_address	semaphore

full_address *full_address_ptr
CAN_do_task_keyword (int) (*intfunptr)()
(void) (*voidfunptr)()
intfunptr set_of_sites_func
voidfunptr connectivity_func:
voidfunptr distribution_func:
voidfunptr coordinate_func:
voidfunptr inverse_coordinate_func:

Exported Variables:

int CAN_nlattices	int CAN_nfields
int CAN_nsets	int CAN_nmaps
int CAN_nrandoms	int CAN_nthreads
int CAN_my_thread	int CAN_stack_size
site *HOME	site NOWHERE

grid NOGRID

CANOPY Limits:

MAXPARAMETERS (57)
MAXFIELDS (200)
MAXLATTICES (10)
MAXSETS (200)
MAXCLUSTERS (20)
MAXPAIRS (20)
MAXFILES (5)
MAXMAPS (20)
MAXGENERATORS (5)
MAX_NODES (630)
MAXARGS (10)
MAXALLOWED (2048)
CAN_ATOMIC_GATHER (512)

CANOPY Keywords:

PASS	FUNCTION
SUM_REAL	INTEGRATE
SUM_INTEGER	SUM_DOUBLE
MAX_REAL	MIN_REAL
MAX_INTEGER	MIN_INTEGER
MAX_DOUBLE	MIN_DOUBLE
TAG_MAX_REAL	TAG_MAX_INTEGER
TAG_MAX_DOUBLE	END
STREAM_PER_SITE	STREAM_PER_NODE

READ WRITE APPEND

Declaration Routines:

```
grid arbitrary_grid
     (int nsites, int ndim, int ndir,
      intptr lower_limits, intptr upper_limits,
      intptr other_params,
      distribution_func dist_func,
      coordinate_func c_func,
      inverse_coordinate_func icfunc,
      connectivity_func conn_func);
field site_field(grid g, int nbytes);
field link_field(grid g, int nbytes);
void overlap_fields(int n, field *list);
void cluster_fields(int n, field *list);
set set_of_sites(grid g, set_of_sites_func func);
set redefine_set_of_sites
     (grid g, set_of_sites_func func,
      set set_to_change);
map define_map
     (grid domain, grid range,
      intfunptr mapfunc);
map compose_map
     (map mid_to_range, map domain_to_mid);
int make_random_generator
     (voidfunptr random_func, int type,
      int number_to_make, int seed);
void declare_lalloc_sizes
     (int do_task_size, int control_size);
void complete_definitions();
```

Data Management:

```
void reset_lalloc();
void broadcast(voidptr object, int nbytes);
void intcpy(voidptr dest, voidptr src, int words);
```

do_task Routines:

```
void do_task
     (voidfunptr task, set s, ..., END);
void do_task_n_times
     (voidfunptr task, set s,
      int ntimes, ..., END);
void do_task_on_inverse_image
     (voidfunptr task, map m, ..., END);
void do_task_on_inverse_image_set
     (voidfunptr task, map m,
      set s, ..., END);
```

A complete triplet is of the form:

<keyword>, <address>, <length in bytes>

Coordinates:

```
void get_coordinates(site *s, intptr coords);
void get_coordinates_at_dir
                    (direction dir, intptr coords);
void get_coordinates_at_path(path p, int *coords);
void sprintf_site_coordinates(char *ss, site *s);
```

Site Management:

```
site site_at_coordinates(grid g, intptr coords);
site site_at_dir(direction dir);
site site_at_path(path p);
site move_site(site *startsite, direction dir);
site move_site_by_path(site *startsite, path p);
logical is_same_site(site *s1, site *s2);
grid grid_supporting_site(site *s);
site image_of_site(map m, site *s);
site *inverse_image_of_site(map m, site *s);
```

Path Routines:

```
int make_path(path p, ...);
int extend_path(path p, direction dir);
int concat_path(path dest, path source);
int copy_path(path dest, path source);
int path_length(path p);
```

Field File Routines:

```
void open_field_file(char *filename, int rwmode);
void close_field_file(char *filename);
void write_field(char *filename, field f);
void read_field(char *filename, field f);
void read_slice_of_field
     (char *filename, field f, intfunptr mapfunc);
```

Field Pointer Routines:

Access by site, direction or path:

Routines have _at_dir *and* _at_path *variations substituting* direction dir *or* path p *for* site *s:

```
voidptr field_pointer(field f, site *s);
void put_field(field f, site *s, voidptr object);
voidptr link_field_pointer
                    (field f, direction link, site *s);
void put_link_field(field f, direction link,
                    site *s, voidptr object);
void synchronize(site *s);
sync_address sync_word(site *s);
voidptr sync_field_pointer (field f, site *s);
field_address address_of_field (field f, site *s);
field_address address_of_link_field
                    (field f, direction link, site *s);
```

Direct use of field address:

```
voidptr field_pointer_from_address
     (field_address *where);
voidptr sync_field_pointer_from_address
     (field_address *where, sync_address *sync);
void put_field_at_field_address
     (field_address *where, voidptr object);
int length_of_field_address_field
     (field_address *where);
```

Information Routines:

```
intptr grid_lower_bounds(grid g);
intptr grid_upper_bounds(grid g);
intptr grid_parameters(grid g);
int number_of_directions_of_grid(grid g);
int number_of_dimensions_of_grid(grid g);
int number_of_fields_on_grid(grid g);
int number_of_sets_on_grid(grid g);
grid grid_supporting_field(field f);
int field_length(field f);
grid grid_supporting_set(set s);
int number_of_sites_in_set(set s);
int number_of_levels_in_set(set s);
intptr nsites_at_each_level(set s);
int level_of_site_in_set(set s, site *ss);
grid domain_grid_of_map(map m);
grid range_grid_of_map(map m);
map *maps_connecting_grids(grid domain, range);
logical is_same_map(map m1, map m2);
```

User-Supplied Routines:

```
void control(int argc, char **argv, char **envp);
void <task_routine>(...);
int <set_of_sites_function>(grid g, intptr coords);
logical <mapfunctionname>
        (intptr incoords, intptr outcoords);
void <random_func> (random_generator_area *a);
void <random_generator>
     (int number_to_make,
      queue_struct *queue, voidptr state);
void <random_initialize>
     (int seed, int stream, voidptr state);
void <distribution_function>
     (grid g, int serial, intptr node, intptr posit);
void <coordinate_function>
     (grid g, int serial, intptr coords);
void <inverse_coordinate_function>
     (grid g, intptr coords, intptr serial);
void <connectivity_function>
     (grid g, intptr coords, site *site_struct);
```

Multithread Control:

```
void multithread(int nthreads, int stack_size);
void multithread_enable();
void multithread_disable();
void print_multithread_stats();
void multithread_begin_vertical();
void multithread_end_vertical();
void multithread_begin_nocopy();
void multithread_end_nocopy();
```

IEEE Precision Control:

```
void set_default_floating_mode();
void set_floating_mode_to_environment();
int get_current_floating_mode();
void print_current_floating_mode();
int set_current_floating_mode(int flags);
```

Random Numbers Routines:

```
float random();
float multi_random(int generator);
```

CHIP Routines (<chip.h>)

```
void remote_read
     (full_address *add, int len, voidptr object);
void remote_read_and_keep
     (full_address *add, int len, voidptr object);
void remote_read_more
     (full_address *add, int len, voidptr object);
void remote_read_and_close
     (full_address *add, int len, voidptr object);
void remote_write
     (full_address *add, int len, voidptr object);
void remote_write_and_keep
     (full_address *add, int len, voidptr object);
void remote_write_more
     (full_address *add, int len, voidptr object);
void remote_write_and_close
     (full_address *add, int len, voidptr object);

void full_address_from_local_address
     (voidptr local_address);
void full_address_on_another_node
     (voidptr address, int node_number);

void yderr(char *errstring);
void do_on_all_nodes(voidfunptr task, ..., END);
logical init_resource(full_address *resource);
logical lock_resource(full_address *resource);
void wait_for_resource(full_address *resource);
void free_resource(full_address *resource);
```

CHIP Exported Variables:

```
int CAN_number_of_nodes;
int CAN_number_of_this_node;
```

Gridlib (<grid.h>)

Defined in <grid.h>

X	MINUS_X
Y	MINUS_Y
Z	MINUS_Z
T	MINUS_T

```
grid periodic_linear_grid(int x);
grid periodic_square_grid(int x, y);
grid periodic_cubic_grid(int x, y, z);
grid periodic_hypercubic_grid(int x, y, z, t);
grid chunky_periodic_square_grid(int x, y);
grid chunky_periodic_cubic_grid(int x, y, z);
grid chunky_periodic_hypercubic_grid(int x, y, z, t);
grid periodic_grid(int ndims, intptr size);
grid chunky_periodic_grid(int ndims, intptr size);
```

Ranlib (<random.h>)

```
void bad_random();
void dual_random();
```

FFTlib (<fft.h>)

```
void FFT_setup(grid g);
void FFT(field f);
void Inverse_FFT(field f);
void FFT_some_directions
        (field f, logical *fft_this_direction);
void Inverse_FFT_some_directions
        (field f, logical *fft_this_direction);
void FFT_double(field f);
void Inverse_FFT_double(field f);
void FFT_double_some_directions(...);
void Inverse_FFT_double_some_directions(...);
```

Setlib (<set.h>)

```
int red_func(grid lattice, int *coords);
int black_func(grid lattice, int *coords);
int br_func(grid lattice, int *coords);
int rb_func(grid lattice, int *coords);
int hyperl_func(grid lattice, int *coords);
int hyperu_func(grid lattice, int *coords);
int spherel_func(grid lattice, int *coords);
int sphereu_func(grid lattice, int *coords);
```

Promptlib (<prompt.h>)

```
double prompt_double(char *prompt);
float prompt_float(char *prompt);
int prompt_int(char *prompt);
void prompt_string(char *prompt, char *ss);
void prompt_word(char *prompt, char *ss);

double prompt_double_default
    (char *prompt, double default);
float prompt_float_default
    (char *prompt, float default);
int prompt_int_default
    (char *prompt, int default);
void prompt_string_default
    (char *prompt, char *default, char *ss);
void prompt_word_default
    (char *prompt, char *default, char *ss);

double prompt_double_range
    (char *prompt, double low, double high);
float prompt_float_range
    (char *prompt, float low, float high);
int prompt_int_range
    (char *prompt, int low, int high);

double prompt_double_range_default
    (char *prompt, double default,
     double low, double high);
float prompt_float_range_default
    (char *prompt, float default,
     float low, float high);
int prompt_int_range_default
    (char *prompt, int default,
     int low, int high);

int prompt_double_list
    (char *prompt, int max, double *list);
int prompt_float_list
    (char *prompt, int max, float *list);
int prompt_int_list
    (char *prompt, int max, int *list);

void prompt_scanf(int expected, char *prompt,
                  char *format, ...);
```

Cmplxlib (<cmplx.h>)

Complex Number Types

```
typedef struct {
  float real;
  float imag;
} complex;

typedef struct {
  double real;
  double imag;
} double_complex;
```

Complex Functions:

```
float cabs(complex *a);
float cabs_sq(complex *a);
float carg(complex *a);
complex cmplx(float r, float i);
complex conjg(complex *a);
complex cadd(complex *a, complex *b);
complex cdiv(complex *a, complex *b);
complex cmul(complex *a, complex *b);
complex csub(complex *a, complex *b);
complex csin(complex *a);
complex ccos(complex *a);
complex ctan(complex *a);
complex casin(complex *a);
complex cacos(complex *a);
complex catan(complex *a);
complex csinh(complex *a);
complex ccosh(complex *a);
complex ctanh(complex *a);
complex casinh(complex *a);
complex cacosh(complex *a);
complex catanh(complex *a);
complex cexp(complex *a);
complex clog(complex *a);
complex ce_itheta(float theta);
complex evaluate_complex_polynomial
    (int order, complex *coef, complex *val);
complex polish_complex_root
    (int order, complex *coef, *root, int nval);
int solve_complex_quadratic
    (complex *coef, complex *root);
int solve_complex_cubic
    (complex *coef, complex *root);
```

Complex Macros:

CONJG(a,b)	b = congj(a)
CADD(a,b,c)	c = a + b
CSUM(a,b)	a += b
CSUB(a,b,c)	c = a − b
CMUL(a,b,c)	c = a * b
CDIV(a,b,c)	c = a / b
CMUL_J	c = a * conjg(b)
CMULJ_	c = conjg(a) * b
CMULJJ	c = conjg(a * b)
CNEGATE(a,b)	b = −a
CMUL_I(a,b)	b = ia
CMUL_MINUS_I(a,b)	b = -ia
CMULREAL(a,f,c)	c = fa
CDIVREAL(a,f,c)	c = a/f

Double Complex Functions:

```
double dcabs(double_complex *a);
double dcabs_sq(double_complex *a);
double dcarg(double_complex *a);
double_complex dcmplx(double r, double i);
double_complex dconjg(double_complex *a);
double_complex dcadd(double_complex *a, *b);
double_complex dcdiv(double_complex *a, *b);
double_complex dcmul(double_complex *a, *b);
double_complex dcsub(double_complex *a, *b);
double_complex dcsin(double_complex *a);
double_complex dccos(double_complex *a);
double_complex dctan(double_complex *a);
double_complex dcasin(double_complex *a);
double_complex dcacos(double_complex *a);
double_complex dcatan(double_complex *a);
double_complex dcsinh(double_complex *a);
double_complex dccosh(double_complex *a);
double_complex dctanh(double_complex *a);
double_complex dcasinh(double_complex *a);
double_complex dcacosh(double_complex *a);
double_complex dcatanh(double_complex *a);
double_complex dcexp(double_complex *a);
double_complex dclog(double_complex *a);
double_complex dce_itheta(double theta);
double_complex evaluate_double_complex_polynomial
    (int order, double_complex *coef, *val);
double_complex polish_double_complex_root
    (int order, double_complex *coef, *root, int nval);
int solve_double_complex_quadratic
    (double_complex *coef, *root);
int solve_double_complex_cubic
    (double_complex *coef, *root);
```