Using Cilk for Parallel Computation in MATLAB

by

Jeremy Sawicki

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author .....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by .....
Charles E. Leiserson
Professor
Thesis Supervisor

Certified by .....
Bruce M. Maggs
Visiting Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Using Cilk for Parallel Computation in MATLAB

by

Jeremy Sawicki

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design and implementation of a system which integrates Cilk, a language for multithreaded parallel programming, with MATLAB, a software package for scientific computing, thereby enabling parallel computation to be used within MATLAB. The system includes a modified version of the Cilk runtime system and a mechanism for sharing it among separately-compiled Cilk functions used within MATLAB. Several example functions and user documentation are provided.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Thesis Supervisor: Bruce M. Maggs
Title: Visiting Scientist

Acknowledgments

I would like to thank Charles Leiserson and Bruce Maggs for their advice and guidance from the inception of this project to its completion. I would also like to thank members of the Cilk group for their assistance, and particularly Harald Prokop for getting me started with the prototype Cilk/MATLAB code and for answering questions along the way.

This work built upon a prototype system developed by Alberto Medina which demonstrated the feasibility of calling Cilk code from MATLAB. The provided example functions are based on examples from the Cilk 5.2 distribution written by Matteo Frigo and Harald Prokop.

As this thesis represents the culmination of my MIT career, I would like to thank my family for the support they have provided over the years, and my friends who have shared this exciting time with me.

Contents

1 Introduction **5**

2 System Design **7**

2.1 Design Goals 7

2.2 MATLAB Overview 8

2.3 Cilk Overview 8

2.4 Runtime System Modifications 9

2.5 Sharing the Runtime System 10

2.6 Cilk Compile-Time Options 13

3 Example Functions **15**

3.1 Shared Runtime System Examples 15

3.2 A Standalone Example 17

4 User Documentation **19**

4.1 Overview 19

4.2 Installation 20

4.3 Operating the Cilk/MATLAB System 21

4.4 Writing a Cilk MEX-file 22

5 The `makestubs` Tool **24**

6 Conclusion **27**

Bibliography **30**

Chapter 1

Introduction

Parallelism has played an important role recently in the quest to increase the performance of modern computers. Current microprocessor designs rely heavily on executing instructions in parallel to reach their performance objectives. And while great strides continue to be made in microprocessor performance, there are always applications that require more computing power than the current generation of processors is able to provide. One way of overcoming this limitation is to use multiple processors working in parallel.

Writing a program to run on multiple processors poses many new challenges, and to make use of parallelism effectively, powerful new tools are required. The Cilk programming language is one such tool. Developed at the MIT Laboratory for Computer Science, Cilk provides abstractions that make it easy to code parallel algorithms to execute efficiently on a variety of platforms, without worrying about system-dependent details [1, 6, 9].

For many users, parallel computation will not be of much value unless it can be used within a framework of existing tools. In many areas of scientific computation, MATLAB is one such popular tool. It provides numeric computation, visualization tools, a high-level programming language, and toolboxes of application-specific functions, all in an integrated environment [8]. Ideally, a researcher could work within this environment most of the time, implementing only the most performance-critical parts of an application in Cilk.

This thesis describes a system that integrates MATLAB and Cilk, making it possible to call Cilk code from the MATLAB environment easily and efficiently. Chapter 2 provides details on the design of the system, and Chapter 3 describes the example functions provided. User documentation is supplied in Chapter 4. Chapter 5 describes a tool that simplifies the use of the Cilk library version. Concluding remarks and a discussion of possible areas for future work can be found in Chapter 6.

Chapter 2

System Design

This chapter describes the design of the Cilk/MATLAB system. First, Section 2.1 discusses the design goals for the system. Sections 2.2 and 2.3 provide brief introductions to MATLAB and Cilk, and particularly the mechanisms that they provide for interfacing with other systems. Changes to the Cilk runtime system necessary to make it conform to MATLAB's memory management requirements are discussed in Section 2.4. Section 2.5 focuses on the issue of sharing a single copy of the runtime system among all Cilk MATLAB functions. Finally, Section 2.6 comments briefly on a remaining problem due to the existence of several incompatible versions of the Cilk runtime system.

2.1 Design Goals

The basic goal of the Cilk/MATLAB system is to allow Cilk functions to be invoked from MATLAB easily and efficiently—that is, the system should be easy to program for, and it should allow machine resources to be utilized efficiently. The need for efficiency is clear: the reason for using parallelism in the first place is to gain additional performance, and any unnecessary overhead will act to degrade this performance advantage. The philosophy I used to determine what can be expected of the programmer is the following: it is reasonable to require that he or she have some knowledge of how to write a Cilk program, and also of how to write an external

MATLAB function, since these are both fundamental to the use of the system. But beyond these requirements, the system should make as few additional demands on the programmer as possible—it should be no more difficult to program for Cilk and MATLAB together than it is for both separately.

2.2 MATLAB Overview

Much of MATLAB's functionality is exhibited to the user in the form of functions. Built-in functions that perform many common tasks are provided as part of the basic MATLAB system, and users can also extend its behavior with their own functions. These can be written in MATLAB's own high-level language, or can be platform-specific compiled code written in either C or Fortran [7].

MATLAB follows the convention that the implementation of each external function is provided in a separate file. For functions written in MATLAB's scripting language, these are simply text files, given the extension `.m` and referred to as M-files. Compiled C or Fortran functions, referred to as MEX-files, reside in dynamically linked libraries and are given platform-specific extensions such as `.mexsol` for Solaris and `.mexsg` for SGI. While this separation of functions into different files is desirable in general, it has implications for how to efficiently make Cilk code available within the MATLAB environment.

2.3 Cilk Overview

A Cilk program consists of two main parts: the Cilk runtime system and the user's compiled Cilk functions. The runtime system handles tasks such as creating worker threads and scheduling the program for execution on multiple processors. Before any of the user's code can be executed, the runtime system must be initialized. After initialization, control passes to the compiled Cilk code. During the course of the program's execution, there are many interactions between the runtime system and the user's code. For instance, memory allocation in a Cilk program is performed

via calls to the runtime system's memory manager. Also, scheduling decisions are made through a series of cooperative interactions between the runtime system and individual Cilk functions.

There are two standard ways of starting up the Cilk runtime system. When a Cilk program is compiled as a standalone executable, it is automatically started up upon execution and used to invoke the Cilk `main` function. To allow Cilk code to be called from a C program, the Cilk library version is provided, which allows the runtime system to be invoked explicitly. The user first calls `Cilk_init` and then can execute Cilk functions as many times as desired by passing function and argument pointers to `Cilk_run`. The `Cilk_finish` function is used to shut down the runtime system.

2.4 Runtime System Modifications

With the Cilk and MATLAB interfaces as described above, it seems that one could simply use the Cilk library version to call a Cilk function from C, and then compile that C code as a MATLAB MEX-file. While this approach basically works, one problem arises. Sufficiently complex systems tend to perform certain operations in proprietary, and conflicting, ways; in the case of Cilk and MATLAB, the culprit is memory management.

MATLAB requires that MEX-files allocate and free memory using MATLAB-specific functions, rather than the standard `malloc` and `free` C library routines. These specialized functions allow MATLAB to provide features such as automatically freeing allocated memory when a MEX-file terminates, as well as perhaps optimizing memory usage in other ways. Needless to say, neither Cilk user code nor the Cilk runtime system ordinarily allocates memory in this way. Therefore, they need to be modified to do so.

Like MATLAB, the Cilk runtime system performs its own memory management, requiring users to allocate memory via functions such as `Cilk_malloc` and `Cilk_free`. In this case, the custom memory handling is an advantage, because it means

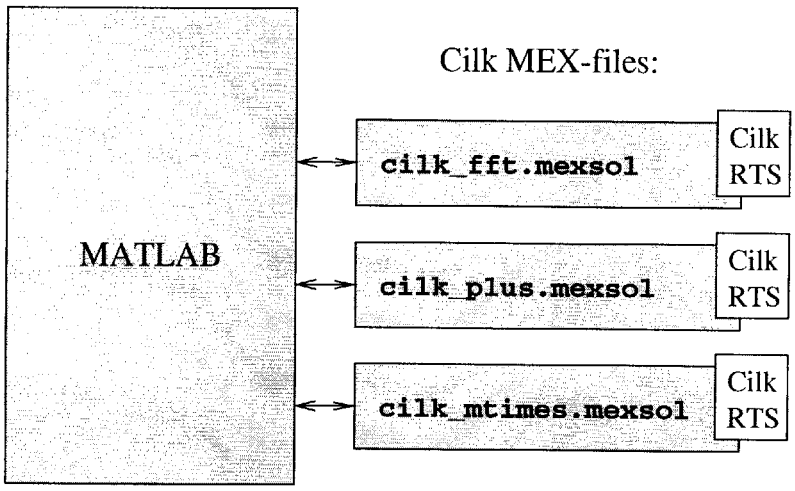

Figure 2-1: Without a mechanism for sharing the Cilk runtime system among several MATLAB functions, many runtime system instances may be in use at once.

that only internal Cilk memory allocation code needs to be modified to work under MATLAB. Since user code already allocates memory through Cilk, it inherits any changes automatically.

The necessary changes are confined to a single source file, `malloc.c`. A modified version is provided that passes all requests to allocate and free memory along to the appropriate MATLAB functions. It can be compiled and linked with the rest of the Cilk runtime system to produce a MATLAB-specific version, and from the programmer's perspective, using it is no different from using the standard Cilk library version.

2.5 Sharing the Runtime System

While the modified version of the Cilk runtime system makes it possible to call a Cilk function from MATLAB, problems begin to arise when there are many such functions. Because MATLAB expects to find each function in a separate file, the most straightforward thing to do is to compile each Cilk MATLAB function separately. But this strategy causes each function to have its own private copy of the runtime system. When a MATLAB application uses several of these functions at once, each

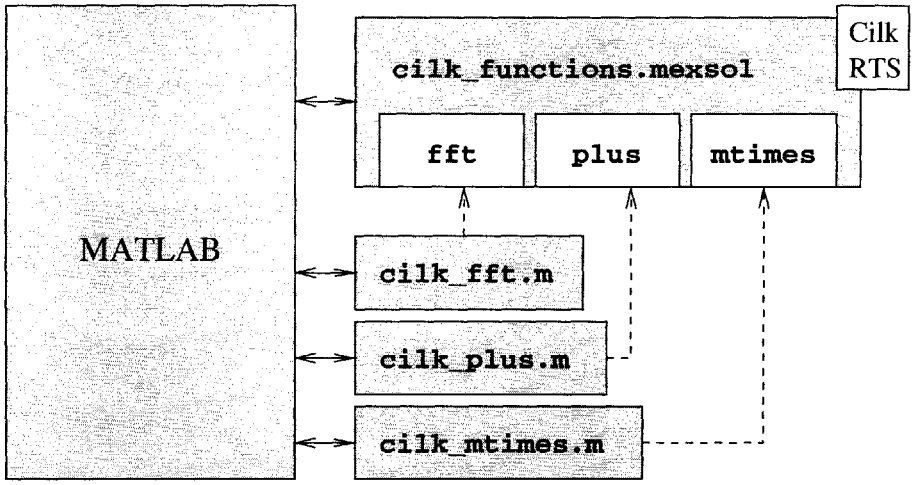

Figure 2-2: Multiple Cilk functions can be compiled into a single MEX-file and still be callable with different names using M-files. This strategy permits sharing of the runtime system, but the functions can no longer be developed independently, a fatal disadvantage.

runtime system instance resides in separate memory, incurs separate startup overhead, manages its own private collection of worker threads, etc. (see Figure 2-1). In contrast, the Cilk library version allows multiple `Cilk_run` calls to be made, executing different Cilk functions, all within the same instance of the runtime system. Ideally, this ability could be passed on to Cilk functions running under MATLAB.

A simple way of allowing multiple Cilk functions to share one copy of the runtime system is to compile them all into a single MEX-file, perhaps using an extra argument to determine which function is to be executed. The user can call these functions in a more natural way by creating M-files with different names that pass their arguments on to the single MEX-file (see Figure 2-2). The disadvantage is that the functions can no longer be developed independently. Ideally, Cilk MEX-files should be compiled and even distributed separately, just like their C and Fortran counterparts, while still benefiting from a shared copy of the runtime system.

The correct approach seems to be to put the Cilk runtime system into one MEX-file, and somehow allow Cilk functions in other MEX-files to be executed under this copy of the runtime system. To see exactly how to achieve this goal, I first had to

determine how isolated MATLAB keeps MEX-files from one another. It turns out that MEX-files can access each other's memory by exchanging pointers encapsulated in MATLAB data types, but unfortunately one MEX-file cannot see the symbols of another, and thus cannot directly call the functions it contains.

At first glance, it seems that all a MEX-file containing a Cilk function would need to do is pass a function pointer, via MATLAB, to the MEX-file containing the Cilk runtime system. Then a single MEX-file would have access to both the runtime system and the function to be executed, and it could simply pass the function pointer along to `Cilk_run`. Unfortunately, the solution is not so simple. The problem is that compiled Cilk code itself contains many calls to functions in the runtime system, for tasks such as scheduling and memory allocation. Making every one of these calls via MATLAB is not really an option. MATLAB is fundamentally a serial program, and it is not prepared to handle calls coming from multiple threads simultaneously. Perhaps if the MEX-file containing the runtime system supplied the calling MEX-file with pointers to all of its functions, these could be used to make the calls directly. This strategy is fairly complex, and it requires substantial changes to compiled Cilk code to allow it to run under MATLAB.

Instead, I found a way of allowing Cilk MEX-files to reference the symbols of the runtime system directly. First, the runtime system is compiled as a shared library, `libcilkmatlab.so`, rather than a statically linked library as is ordinarily done. Shared libraries are accessed via calls such as `dlopen`, `dlsym`, and `dlclose`, which permit an application to look up and call functions that a library contains. The `dlopen` function can also be given a special flag, `RTLD_GLOBAL`, indicating that the symbols in the library should be made available globally—within the entire application. If the Cilk runtime system is accessed using this flag from within a MEX-file, all other MEX-files will then be able to reference Cilk symbols directly—Cilk is effectively imported as a part of MATLAB itself (see Figure 2-3). The advantage is that no changes must be made to the way Cilk user code is compiled.

A word of caution is in order: Importing the symbols of a shared library globally from within a MEX-file is not likely to be supported by The MathWorks. Provided

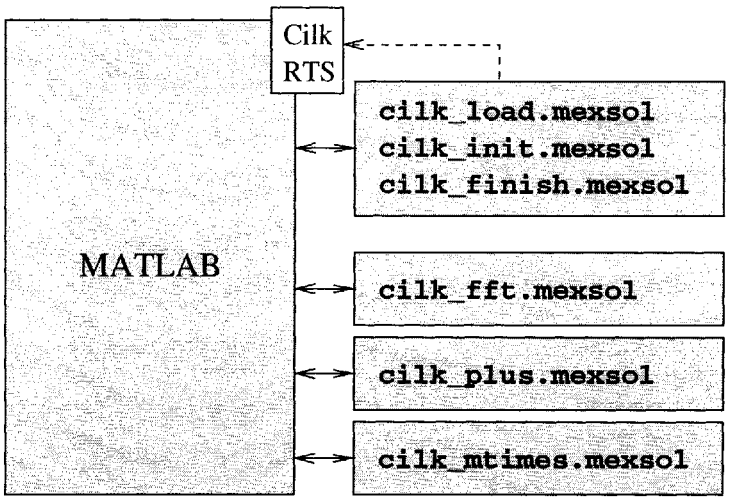

Figure 2-3: The current Cilk/MATLAB system effectively makes the Cilk runtime system a part of MATLAB, so that it is available automatically to any Cilk MEX-file. Auxiliary MEX-files handle loading, initializing, and terminating the runtime system.

that the symbols are named so as to avoid conflicts (all beginning with `Cilk_` for instance), however, there is not likely to be a problem, and I expect that this trick would work on any platform supporting similar functionality to `dlopen`.

Once the Cilk runtime system has been made available to all MEX-files, it is straightforward to write a Cilk function that can be called from MATLAB. The programmer need only conform to the standard interfaces for MATLAB C language MEX-files, and invoke the Cilk function using `Cilk_run`. Only two things must be done differently to take advantage of the shared runtime system. First, there is no need to call `Cilk_init` and `Cilk_finish`. The user controls when to start up and terminate the shared runtime system via auxiliary MEX-files provided for that purpose. Second, the file should of course not be linked with the runtime system, if the shared copy is to be used.

2.6 Cilk Compile-Time Options

One additional difficulty arises due to the fact that Cilk supports several options which must be specified at compile time. These options control whether or not code

for debugging, timing, and statistics collection is compiled into a user's program. Currently, the same set of options must be supplied when compiling both the runtime system and each of the Cilk functions it will be used to execute. This presents a problem when trying to share a single copy of the runtime system among many different Cilk functions, which may be developed or distributed separately. One set of options must be chosen for the shared runtime system, and only Cilk functions which conform to this choice will be able to take advantage of it.

Unfortunately, no easy solution exists to this problem. The possibility has been discussed of a future version of Cilk providing a unified runtime system that can support user code compiled with any set of options. Until these changes are made, the only alternative is for functions compiled with different options to use different copies of the runtime system. To allow individual functions to deviate from the set of options in use by the main Cilk/MATLAB system, a statically linked version of the MATLAB-modified runtime system is also provided. Each of these functions must have its own copy of the runtime system—the method for sharing the runtime system does not permit multiple versions to be shared simultaneously.

Chapter 3

Example Functions

This chapter describes the example functions provided as part of the Cilk/MATLAB distribution. These examples demonstrate the mechanics of writing a Cilk MEX-file, as well as some of the issues that arise when converting existing Cilk code to run under MATLAB.

3.1 Shared Runtime System Examples

The following examples are designed to work with MATLAB's shared copy of the Cilk runtime system; thus, they do not include calls to `Cilk_init` or `Cilk_finish`, only `Cilk_run`.

Matrix Addition

`CILK_PLUS(X,Y)`

This function is a simple parallel matrix addition example. It handles both real and complex inputs, matrices and N-dimensional arrays, and can add a scalar to an array or matrix as well as adding two identically sized matrices.

Remarks

This function is a good example of a MEX-file that attempts to mimic all of the different behaviors of a built-in MATLAB function. It checks its arguments carefully, and falls back gracefully to MATLAB's built-in PLUS function if it receives any input that it is not prepared to handle.

Matrix Multiplication

```
CILK_MTIMES(X,Y)
```

This function implements a divide-and-conquer matrix multiplication algorithm using a blocked matrix representation. It is based on `rectmul.cilk` from the standard Cilk distribution, modified to handle complex as well as real matrices. Multiplying a scalar by a matrix or N-dimensional array is also supported.

Remarks

This example illustrates the problem of differing data representations commonly encountered when modifying existing code for use with MATLAB. MATLAB stores matrices in column-major order, so to multiply them with the `rectmul` code, they must be converted between this representation and the blocked row-major representation that the algorithm expects.

Fast Fourier Transform[1]

```
CILK_FFT(X)
CILK_FFT(X,N)
CILK_FFT(X,[],DIM)
CILK_FFT(X,N,DIM)
```

[1]This FFT code seems to be several times slower than MATLAB's built-in version on some inputs. For a fast, portable FFT implementation, consider FFTW [4, 5], which includes a parallel version written in Cilk. FFTW comes with a MATLAB MEX-file designed for calling the serial version, which I was easily able to modify to execute the Cilk version using the shared copy of the runtime system.

This function is a MATLAB version of the Cooley-Tukey [3] one-dimensional Fast Fourier Transform code found in `fft.cilk` from the standard Cilk distribution. It implements the same semantics as MATLAB's `FFT` command, including the N-point FFT and computing the FFT of each column of a matrix or across an arbitrary dimension of an N-dimensional array.

Remarks

MATLAB stores the real and complex parts of a matrix in separate arrays. This FFT code expects both parts of a number to be stored together, so some conversion is necessary. Extra code "deinterlaces" the input if multiple FFTs are being performed across any dimension of an array other than the first.

Overloading Functions

The three example functions described above are designed to be able to replace their built-in MATLAB counterparts. MATLAB allows a built-in function to be overloaded by placing an M-file or MEX-file of the same name in a special directory called `@double` (for functions operating on matrices of `doubles`) somewhere in the MATLAB path. The example Cilk MEX-files themselves should not be placed in this directory, because then they will be called even if the Cilk runtime system has not been initialized. Instead, M-files should be created which test the `CILK_RUNNING` global MATLAB variable and only call the Cilk versions if Cilk has been initialized. Three M-files which do this for the above examples, `plus.m`, `mtimes.m`, and `fft.m`, are provided.

3.2 A Standalone Example

A Cilk MEX-file can be compiled with its own private copy of the Cilk runtime system. In this case, the MEX-file must handle starting up and shutting down the runtime system. The following function provides one example of how this can be done.

Fibonacci Numbers

`FIB(N)`

This function calculates the Nth Fibonacci number using a slow, doubly recursive algorithm.

Remarks

The key feature of this example is that it handles initializing and terminating the Cilk runtime system transparently to the user. It keeps track of whether or not it has been called before, and on the first call only, runs `Cilk_init`. It also registers an exit function using MATLAB's `mexAtExit` call, so that when the MEX-file is cleared from memory or MATLAB exits, it has the chance to call `Cilk_finish`.

Chapter 4

User Documentation

This chapter is a users and programmers guide for the Cilk/MATLAB system. Section 4.1 gives an overview of the system, and Section 4.2 covers compilation and installation. The tools provided to manage the system within MATLAB are described in Section 4.3, and Section 4.4 explains what must be done to create a Cilk MATLAB function.

4.1 Overview

The purpose of the Cilk/MATLAB system is to allow Cilk code to be used within external MATLAB functions (MEX-files). The basic approach is to write a C language MEX-file that uses the Cilk library version to invoke Cilk code from C. Because MATLAB expects MEX-files to allocate memory via its memory manager, a MATLAB version of the Cilk runtime system with modified memory management code is provided.

A mechanism for sharing a single copy of the Cilk runtime system among all Cilk MEX-files is also provided. The runtime system is compiled as a shared library, and an auxiliary MEX-file loads this library as part of MATLAB in such a way that other MEX-files can reference its symbols. Additional commands are provided which allow the user to initialize and shut down the shared runtime system within MATLAB.

The Cilk/MATLAB system is based on Cilk version 5.2. It was developed using

MATLAB 5.0, though it should work with any MATLAB 5.x release. Currently it has only been tested on Solaris.

4.2 Installation

To install the Cilk/MATLAB system, you will first need to obtain both the base Cilk 5.2 distribution and the Cilk/MATLAB archive. These can be downloaded from the Cilk web page at `http://supertech.lcs.mit.edu/cilk/`. Note that Cilk requires `gcc` as well as GNU's `make` utility. You will also need to have your system set up to compile MEX-files using MATLAB's `mex` command.

To compile and install the Cilk/MATLAB system, perform the following steps:

1. Unpack the Cilk 5.2 archive by running `gunzip <filename>.tar.gz` and `tar xvf <filename>.tar`. Below, `<cilk dir>` refers to the top-level directory of the Cilk distribution (usually named `cilk`).

2. Unpack the Cilk/MATLAB archive inside the Cilk directory, as `<cilk dir>/matlab`.

3. Edit the file `<cilk dir>/matlab/Makefile.matlab.common`, changing `MATLAB_INCLUDE_DIR` to point to the directory where MATLAB's include files reside on your system.

4. In `<cilk dir>/matlab`, run `./install` to modify the necessary files from the Cilk distribution to compile a MATLAB-specific version of the Cilk runtime system.

5. In `<cilk dir>`, run `./configure --with-os-threads --enable-matlab` and `make` to compile the runtime system. Other Cilk compile time options can be supplied to `configure` as well.

6. In `<cilk dir>/matlab`, run `make` to compile the shared library version of the runtime system, the auxiliary MEX-files needed to operate it, and the example functions.

7. Copy the following M-files and MEX-files into a directory in MATLAB's search path:

   - `<cilk dir>/matlab/cilkengine/*.mex*`
   - `<cilk dir>/matlab/examples/*.mex*`
   - `<cilk dir>/matlab/M-files/*.m`

8. Copy `<cilk dir>/matlab/lib/libcilkmatlab.so` to wherever you would like it to reside, and edit `cilk_load.m` to point to that location.

9. If you wish to overload MATLAB's built-in `FFT`, `PLUS`, and `MTIMES` functions with the Cilk versions, create a directory called `@double` somewhere in MATLAB's search path, and copy `<cilk dir>/matlab/M-files/@double/*.m` into that directory.

4.3 Operating the Cilk/MATLAB System

The following functions[1] are provided for controlling MATLAB's shared copy of the Cilk runtime system (see Figure 4-1):

`cilk_load` This command uses `dlopen` to dynamically link to the shared copy of the runtime system, making it available to all MEX-files. Calling this function is not strictly necessary, because `cilk_init` calls it automatically if the runtime system has not yet been loaded.

`cilk_init` This command is the MATLAB equivalent of the `Cilk_init` library version call, which initializes the runtime system and prepares it to execute Cilk functions. Runtime system options may be supplied, e.g. `cilk_init('-nproc 4')`.

[1] These functions are implemented as M-files. The actual work is done by the MEX-files `cilk_load_internal`, `cilk_init_internal`, and `cilk_finish_internal`.

```
>> a=rand(1000,1000);
>> b=rand(1000,1000);
>> cilk_load
>> cilk_init('-nproc 1')
>> tic; c=cilk_mtimes(a,b); toc

elapsed_time =

    18.0821

>> cilk_finish
>> cilk_init('-nproc 2')
>> tic; c=cilk_mtimes(a,b); toc

elapsed_time =

    9.0914

>> cilk_finish
>> cilk_clear
>>
```

Figure 4-1: An example of how to execute Cilk MEX-files using MATLAB's shared copy of the Cilk runtime system.

`cilk_finish` This command is the MATLAB equivalent of the `Cilk_finish` library version call, which shuts down the Cilk runtime system. Calling `cilk_finish` is useful if no more calls to Cilk MEX-files will be made, or if the runtime system needs to be reinitialized with a different set of options.

`cilk_clear` This command uses `dlclose` to release the shared library. It also clears the Cilk-related MEX-files (`cilk_load_internal`, `cilk_init_internal`, and `cilk_finish_internal`) from memory.

The above functions maintain two global MATLAB variables, `CILK_LOADED` and `CILK_RUNNING`, which can be tested to determine whether or not the shared copy of the Cilk runtime system has been loaded or initialized, respectively.

It is important to load and initialize the shared runtime system before attempting to call a Cilk MEX-file that depends on it. If the Cilk library has not been loaded, MATLAB will reject the MEX-file because it references Cilk symbols that are undefined. If the runtime system has been loaded but not initialized, or if it has been shut down by calling `cilk_finish`, attempting to use a Cilk MEX-file may have unpredictable results.

4.4 Writing a Cilk MEX-file

The reader is assumed to have some familiarity with how to program in Cilk and how to write a MATLAB C language MEX-file. Information on programming in Cilk is

available in [2], and the MATLAB API is described in [7].

A Cilk MEX-file is basically a C MEX-file which calls Cilk code via `Cilk_run`. Thus you must still define the standard `mexFunction` entry point for MATLAB, which should be a C function. You can use all of the standard MATLAB API calls to get information about arguments, allocate space for output matrices, etc. It is recommended that all calls to MATLAB API functions be made from C before invoking a Cilk function with `Cilk_run`. While it should be possible to make these calls from Cilk code, care must be taken not to have more than one call outstanding at a time, because MATLAB is not designed to support multithreaded MEX-files.

If a MEX-file will be using the shared copy of the Cilk runtime system, it should not include calls to `Cilk_init` or `Cilk_finish`—the user bears responsibility for initializing the runtime system in MATLAB prior to calling any Cilk MEX-files. If the MEX-file will have its own private copy of the runtime system, it must handle making these calls itself. One convenient way of initializing and terminating the runtime system transparently to the user is demonstrated in the `fib` example function.

Cilk MATLAB source files are compiled like any other Cilk files, and linked using MATLAB's `mex` command. To use a private copy of the runtime system, `cilk-matlab-rts.a` and `cilk-lib.a` should be specified as arguments to `mex`. These libraries are both included in the shared Cilk runtime system, so neither should be linked with a MEX-file that uses the shared copy. Because Cilk uses `gcc`, it is necessary to link in `libgcc.a` in both cases. To get everything set up properly, you may find it convenient to copy the `Makefile` from the `examples` directory to use as a starting point. Remember to change `CILK_DIR` to point to the location where Cilk is installed on your system.

Chapter 5

The makestubs Tool

The Cilk library version allows a Cilk function to be invoked from C via `Cilk_run`, but the function must have a specific type—it must take a single argument of type `void *` and return an `int`. While it is possible to use this mechanism to call a Cilk function with arbitrary argument types, doing so can be cumbersome. Typically, a C structure is created to contain all of the desired arguments, and a pointer to the structure is passed via `Cilk_run`. Extra code is required on both the C side to copy the arguments into the structure, and on the Cilk side to extract the arguments and pass them to the Cilk function being called. Also, if the Cilk function returns a value needed by the calling C code (of some type other than `int`), it must be passed back as another field in the structure.

The `makestubs` program automates the process of writing wrapper code to call an arbitrary Cilk function from C. It reads in an input file containing one or more Cilk function prototypes and generates the code necessary to call these functions easily from C via the Cilk library version. For each Cilk function, it creates a structure whose fields correspond to the arguments of the function. A C wrapper function with the same argument and return types as the Cilk function is created, having the same name with `_lib` appended. This function copies its arguments into the structure and then calls `Cilk_run`. The Cilk stub function called by `Cilk_run` is also created. Its job is to call the original Cilk function using the contents of the structure as arguments. If the Cilk function has a non-`void` return type, these wrapper functions

```
cilk double * foo(double *A, double *B, int n);
```

(a)

```
double * foo_lib(double *A, double *B, int n);
```

(b)

```
#include <cilk.h>

typedef struct foo_argstruct {
  double * A;
  double * B;
  int n;
  double * _retval;
} foo_argstruct;

cilk int foo_stub(foo_argstruct *arg)
{
  arg->_retval =
    spawn foo(arg->A, arg->B, arg->n);
  sync;

  return 0;
}
```

```
double * foo_lib(double * A,
                 double * B,
                 int n)
{
  foo_argstruct arg;

  arg.A = A;
  arg.B = B;
  arg.n = n;

  Cilk_run(foo_stub, &arg);

  return arg._retval;
}
```

(c)

Figure 5-1: **(a)** `foo.l`: An example Cilk function prototype, given as input to `makestubs.pl`. **(b)** `foo.l.h`: The corresponding C wrapper function prototype, used to invoke the Cilk function from C code. **(c)** `foo.l.cilk`: The wrapper functions themselves.

also handle passing the return value back to the calling C code. A value of type `int` is returned directly by `Cilk_run`; other types are placed in an extra field of the argument structure.

In addition to the Cilk source file containing all of the wrapper functions and structure definitions, a C header file is created which contains prototypes for the C wrapper functions. This file can be included by any C source file needing to execute the Cilk functions.

Figure 5-1 shows sample input and output files of the `makestubs` tool. For other examples of its use, see the `cilk_plus` and `cilk_mtimes` functions in the `examples` directory of the Cilk/MATLAB distribution.

The `makestubs` program is implemented as a Perl script, `makestubs.pl`. On the command line it takes the names of files containing Cilk function prototypes, typically given the extension `.l`. For each file listed, it produces the two output files described above, given the extensions `.l.cilk` and `.l.h`.

The parsing job that `makestubs` does is fairly simple. It assumes that each argument in a function prototype consists strictly of a type followed by an identifier. The type can consist of any string of characters, but the identifier must come last. Thus all arguments in the function prototype must be given names, and syntax such as `int a[]` and function pointer types are not handled properly.

Chapter 6

Conclusion

In this thesis I have presented a system that permits Cilk code to be used within the MATLAB environment easily and efficiently. It includes the following components:

- A modified version of the Cilk runtime system that allocates memory via MATLAB's memory manager in order to be able to run within a MEX-file
- A dynamically linked library version of the runtime system and associated tools that allow it to be shared by all Cilk MEX-files
- Several example functions which demonstrate the use of the Cilk/MATLAB system
- User and programmer documentation

From the programmer's point of view, writing a Cilk MEX-file is no more difficult than using the library version of Cilk to call a Cilk function from C, and compiling that C code as a C language MEX-file. Within MATLAB, a user need only make a single call to `cilk_init` to have access to all MEX-files that use MATLAB's shared Cilk runtime system.

Performance of the system seems to be good. The overhead of calling a Cilk MEX-file from MATLAB is small, and test data indicates that Cilk code running under MATLAB exhibits the same type of speedup on multiple processors as is typical in standalone Cilk applications (see Figure 6-1).

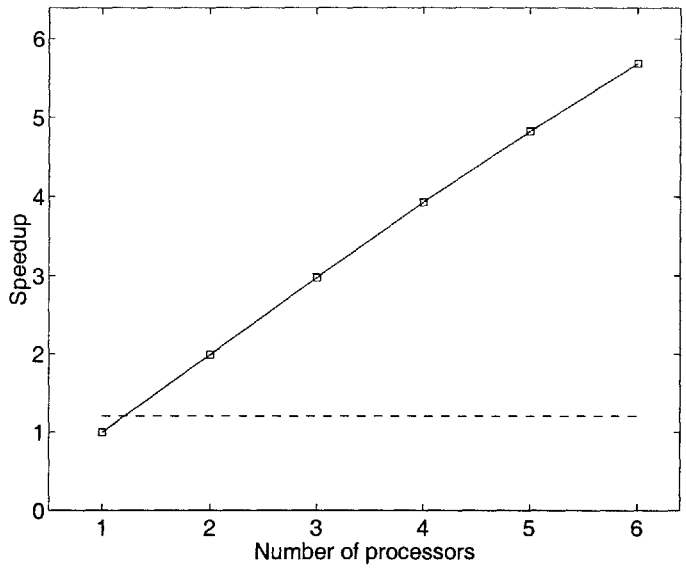

Figure 6-1: This graph compares execution times to multiply two 1000×1000 matrices with the `cilk_mtimes` example using from one to six processors. Speedup is defined as the ratio between the execution times on 1 and n processors. For comparison, the dashed line represents MATLAB's built-in serial matrix multiplication code.

The Cilk/MATLAB system has been made available for download from the Cilk web page, and can be used by anyone wishing to use parallel computation from the MATLAB environment.

Further Work

There are a number of opportunities for further work on the Cilk/MATLAB system and related parts of Cilk. Several of these are described below.

Cilk can collect various statistics during execution that are useful when developing a program, but currently the library version of Cilk does not support this feature. Thus, the statistics cannot be accessed from MATLAB. MATLAB would be more useful as a development environment for Cilk programs if these statistics were available. While a MATLAB-specific solution could be developed, it would be better to make statistics available as a standard part of the Cilk library version. Doing so requires defining semantics for how statistics should be tracked across multiple

`Cilk_run` calls—whether they should be reset each time, kept cumulatively, or cleared only when the user requests it, for instance.

The Cilk/MATLAB system has so far only been tested under Solaris. It should be tried on the other platforms supported by both Cilk and MATLAB, and modified if necessary to run on those platforms. A related issue is that the current version of Cilk does not support a library version on some platforms. Making the library version available is a necessary first step to supporting the Cilk/MATLAB system on those platforms.

As mentioned in Section 2.6, one major area for future work on Cilk which affects the Cilk/MATLAB system is the plan to unify all of Cilk's compile time options in a single version of the runtime system. This version will support Cilk code compiled with any set of options, eliminating the need to statically link a Cilk MEX-file with a private copy of the runtime system in order to deviate from the set of options in use by MATLAB's shared copy.

Another area for further research more closely related to the Cilk/MATLAB system involves providing more sample Cilk MEX-files. Perhaps producing a suite of parallel matrix or linear algebra algorithms for MATLAB written in Cilk would be a valuable project.

Bibliography

[1] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[2] *Cilk 5.2 Reference Manual.* Available from `http://supertech.lcs.mit.edu/cilk/`.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, April 1965.

[4] Matteo Frigo and Steven G. Johnson. The FFTW web page. `http://theory.lcs.mit.edu/~fftw`.

[5] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[6] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[7] The MathWorks, Inc. *MATLAB Version 5 Application Program Interface Guide*, December 1996.

[8] MATLAB is a product of The MathWorks, Inc. Information is available at `http://www.mathworks.com/products/matlab/`.

[9] Keith H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.