This handout is meant to serve as a “getting started” document for parallel programming on the HP/Convex Exemplar at NCSA. It briefly describes the architecture and programming environment on the Exemplar, focusing on Exemplar extensions to the C programming language for compiler-assisted data-parallel programming.

Accessing the Exemplar

Telnet to billie.ncsa.uiuc.edu. Those of you who have registered for the course should have an account on the machine. Please let me know at the earliest if this is not so. You should change your default password the first time you log in.

Architecture

The Exemplar SPP-2000 is a distributed shared memory (DSM) machine built around the “hypernode” configuration. Each hypernode consists of up to 16 PA-8000 processors (180 MHz) and 16 Gbytes of memory. While the configuration at NCSA consists of 4 hypernodes, in this course we shall restrict our programming to a single hypernode. Each hypernode is a symmetric shared-memory multiprocessor: the 16 processors are connected through a non-blocking crossbar switch, with a 64-bit wide datapath and 960 MB/second bandwidth. The PA-8000 processor has 10 functional units and is capable of performing up to 4 floating-point operations per clock cycle, resulting in a peak performance of 720 MFLOPS per processor. It has a 2 megabyte (MB) external, direct-mapped cache (1 MB data and 1 MB instruction caches). The cache line is 32 bytes long. Look at http://www.ncsa.uiuc.edu/SCD/Hardware/SPP2000/ for additional details.

Programming and Execution Environment

The Exemplar runs a variant of the HP-UX (a Unix variant) operating system. Standard UNIX programs as well as all the GNU Utilities are available in /usr/ncsa/bin.

All your programs will be written in the C language. To compile these programs for parallel execution you should use the Exemplar C compiler available as /opt/ansic/bin/cc. A sample makefile is provided in my home directory (~vijayk/homework2/) on billie.ncsa.uiuc.edu which contains the compiler options you will need for Homework 2. Look at http://www.ncsa.uiuc.edu/SCD/Hardware/SPP2000/Doc/ for additional details about Exemplar C.

The makefile produces a parallel executable. There are two alternate ways in which the executable can be run with a specific number of threads: by setting the environment variable MP_NUMBER_OF_THREADS, and using the mpa utility. These two methods are demonstrated in the shell commands below:

```
> setenv MP\_NUMBER\_OF\_THREADS 4
> a.out
> mpa -min 4 -max 4 a.out
```
For performance measurements, you will need to use the batch submission system instead of running the program interactively. The NCSA Exemplar runs the job manager lsbatch, which is a load-sharing batch system built on top of the Load Sharing Facility (LSF) system. A complete list of lsbatch commands (each with a man page) can be found in the man page for lsbatch. The commands you will most likely use are bsub (for submitting a job script), bjobs (to query the system about status of submitted jobs and the various queues), and bkill (to kill a process that is running longer than expected). I have provided a sample job submission script in my home directory (~vijayk/homework2) on billie.ncsa.uiuc.edu. Look at http://www.ncsa.uiuc.edu/SCD/Hardware/SPP2000.Doc/ Jobs.html for additional details about running jobs in both interactive and batch modes on the NCSA Exemplar.

Finally, to debug your parallel program and profile its performance, use the cxdb and cxperf utilities. Look at the man pages for details on these utilities. Of course, you should make every effort to ensure that your program is correct (that it runs on a sequential machine, and on one node of the Exemplar) before you try and run it on the parallel machine with multiple threads.

Information Resources

In addition to the URLs provided above, the following are good sources of information about the Convex Exemplar:

- The HP-Convex Exemplar page at Navy Research Laboratory (NRL):
- The Exemplar itself has a very nice X-windows based document browser called pinpoint which contains a searchable database of all documents relevant to the Exemplar.

Exemplar C: Parallelization Pragmas

Exemplar C allows compiler-assisted parallelization of sequential programs. Two modes are supported: automatic parallelization, and explicit parallelization. In the former, the compiler detects loop-based parallelism by conducting aggressive dependence analysis and generates code which distributes loop iterations among program threads. In the latter mode, the programmer associates directives with portions of the code to express parallelism at the loop, task, and region levels. These directives are typically specified using C pragma statements and can both guide (in a non-binding fashion) as well as direct the compiler to generate parallel code. The Exemplar C compiler accepts several directives: look at the Exemplar C and Fortran 77 Programmer’s Guide available from http://www.ncsa.uiuc.edu/SCD/Hardware/SPP2000/Doc/ for details on the various compiler options. Here we focus only on a handful of directives which can be used to express loop-based and region-based parallelism.

The execution model adopted by the compiler is as follows. The program starts off with the number of threads specified by either the MP_NUMBER_OF_THREADS environment variable or the mpa utility. However, only one thread is active and responsible for executing the program, while the remaining threads stay inactive. On encountering a parallel construct (loops, tasks, or regions), some or all of these threads are activated. When execution of the construct completes, all but one of the threads re-enter the inactive state.

The Exemplar C compiler does not directly support data-parallelism, i.e., it does not provide constructs such as DECOMP to specify data distribution and thereby influence loop iteration assignment according to the owner-computes rule. However, data-parallelism can be emulated in one of two ways. The first approach explicitly identifies that all iterations in a loop can be executed in parallel, optionally specifying how the iterations are divided among active threads. The second approach has each thread execute the same code
(region-based parallelism), with each thread selecting which portion of the code it executes. The owner-computes rule can be explicitly coded in the latter case.

The first approach relies on the `loop_parallel` directive which states that all iterations of the loop immediately following the pragma are independent and can be executed in parallel. To ensure that the iterations are in fact independent (particularly with respect to induction variables), the programmer may privatize scalars and statically allocated array variables using the `loop_private` construct. For Homework 2, two variants of the `loop_parallel` will be of interest, used as shown in the following code fragment:

```c
/* compiler strip-mines the loop among available threads */
#pragma _CNX loop_parallel( ivar = j )  
for ( j=0; j< niters; j++ )
  ...

/* compiler cyclically allocates loop iterations, in units of chunk_size among available threads */
#pragma _CNX loop_parallel( ivar = k, chunk_size = m )  
for ( k=0; k< niters; k++ )
  ...
```

The `ivar` attribute of the `loop_parallel` directive specifies the induction variable defining the iteration space. The compiler requires that loop bounds be loop-independent, allowing run-time distribution of the iterations among participating threads. As these example fragments show, the `loop_parallel` directive can be used to emulate the generic `forall` construct described in Lecture 2.

The second approach for emulating data-parallel programs relies on the `parallel` and `end_parallel` directives which demarcate a region of code that is executed in parallel by all the threads. Predicates involving the thread identifier can be used to achieve thread-specific execution. The total number of threads available to the program, and the identifier of the current thread can be queried at run time using the `num_threads()` and `my_thread()` system calls. The following code fragment shows an example of using the `parallel` directive:

```c
/* the region enclosed by parallel and end_parallel directives is executed in parallel by all the threads */
#pragma _CNX parallel
  ...
/* the following emulates owner-computes distribution */
for ( i=0; i< niters; i++ ) {
  if ( OWNER(i) == my_thread() )
    ...
}
#pragma _CNX end_parallel
```

Additional directives are also provided to enforce synchronization between threads using two mechanisms: gates and barriers. Both gates and barriers are special kinds of variables that need to be dynamically allocated (and initialized) and require special functions to use. For example, the following code fragment shows how a barrier variable is declared, allocated, used, and freed.
int nthreads;
barrier_t my_barrier;   /* declaration */
...
nthreads = num_threads();
alloc_barrier(&my_barrier);   /* allocation */
...
#pragma _CNX parallel
...
for ( i=0; i< niter; i++ ) {
    ...
    wait_barrier(&my_barrier, &nthreads);   /* use */
}
#pragma _CNX end_parallel
...
free_barrier(&my_barrier);

For additional details, refer to Chapters 4 and 6 (basic and advanced shared-memory programming) of the Exemplar Programming Guide available using pinpoint.