This assignment has three objectives: first, to become familiar with shared-memory parallel programming using the ANL portable macros; second, to understand the performance characteristics of hardware versus software support for the shared memory programming model; and finally, to analyze the performance implications of VIA-like architectures, which support remote memory write operations.

As with Homework 3, this assignment requires you to conduct measurements on both the Beowulf cluster and the Origin 2000 machine at NCSA. The former supports the ANL macros shared memory model in software (using an abstraction referred to as software distributed shared memory), while the latter supports it directly in hardware. Other than requiring use of different macro definitions on the two machines, your application should otherwise run unmodified on the two platforms. So, you should continue using the Beowulf cluster for all of your program development and debugging work. To run your programs, you will need to use the `run_hlrc` script in `/usr/local/bin`.

To ensure that you start working with your project group as soon as possible, I shall allow you to do this assignment in groups. Everyone in the group will receive the same grade. You are of course free to do the assignment on your own if you so desire.

### Particulars

For this assignment, you will parallelize integer radix sort, a core kernel used by applications in several domains. Radix sort takes as its input a sequence of (unsorted) integers and produces the sorted sequence as output. A simplified version of the algorithm works as follows: assuming that the integer keys are in the range $[0, 2^k]$ (i.e., representable as a $k$-bit number), the algorithm progresses through multiple rounds each of which sort the keys according to an $r$-bit radix, starting from the least significant bits. For this assignment, we assume that $k$ is a multiple of $r$.

The individual sorting steps themselves consist of three phases: (a) the keys are first scanned to count the number that correspond to each radix value; (b) these counts are aggregated to obtain a rank value for each radix; and (c) finally, these rank values are used to determine the position of the keys in the sorted array.

To help you get started, I have provided a sequential version of the radix sort program accessible on `bm.scs.cs.nyu.edu` at the following location: `/home/vijayk/Homework4/radix.c`. Instructions about files on the Origin will be posted on the class mailing list in a day or two.

Run `radix` without any arguments to see what command line arguments it expects: $r$: the number of radix bits, $k$: the number of key bits (required to be a multiple of $r$), and $n$: the number of keys that need to be sorted. The `radix` program randomly generates the keys and sorts them in $k/r$ rounds. The comments in the code should guide you on the parts that need to be modified for this assignment. For your measurements, you should run the program with the following arguments:

```
> ./radix 5 30 2000000
```

Given this background, the assignment consists of three parts:

1. **(5 points)** Parallelize the radix program using the ANL macros for shared memory programming.

   The `Homework4` directory on `bm.scs.cs.nyu.edu` also contains a program, `simple.c`, and
the corresponding makefile, showing the use of a subset of the ANL macros. Pay close attention to the comments in this file: the macros on both platforms have been defined assuming extremely stylized usage.

To parallelize the program, perform the following steps:

- Partition the program arrays, `keyArray[*]`, `rankArray`, and `numInBins`, among the processors. A block partitioning of these arrays works well, so please do not try and waste your time trying to come up with anything more elaborate.
- Modify the `slaveSort` routine so that it now works on a separate partition of the key array, accumulating the `densityArray` values into a global `rankArray`. You will also need to modify the part of the code that is responsible for computing the start indices (on different processors), for each radix value.
- Insert appropriate synchronization calls in the code to ensure that the shared memory is updated consistently. This specifically applies to updating the rank array, dividing up the rank range of a particular radix value among the various processors, and finally, moving the keys between the “from” and “to” arrays.

If you search the web, you will find several references to parallel radix sort, including some written using the ANL macros. However, I would recommend that you try and develop your solution on your own: adapting the solutions you find to the supplied sequential program will probably end up taking more time (and not prepare you to answer the rest of the assignment) than just working through the assignment on your own.

2. (5 points) The goal of this part of the assignment is to understand the performance characteristics of hardware- and software-supported shared memory. Specifically, the Origin manages coherence at the granularity of 128-byte cache lines (L2), while the software-based HLRC distribution used on the Linux cluster maintains coherence at the granularity of 4096 bytes (a page).

Tune your program of part 1 to improve performance under each architecture for 2, 4, 8, and 16 processors (the last only on the Origin). You will need to pay attention to whether or not two variables are mapped to the same cache block (this can be controlled by inserting appropriate padding), and also be aware of the “first-touch” default policy for allocating pages to processors. The last is needed to ensure that the software shared memory protocol does not unduly burden any one processor, instead uniformly spreading the coherence overheads. You may also need to make your synchronization operations more parallel: note that the `rankArray` update and its use to infer start indices can be modeled as a prefix-sum operation, for which there are efficient algorithms.

3. (7.5 points) For larger numbers of keys, the dominant cost of parallel execution stems from the communication costs involved in the transfer of keys between the “from” and “to” arrays (in the `slaveSort` routine). Although the communication is implicitly expressed as a simple memory-copy loop, this loop involves different numbers of operations based on whether it is being supported on a hardware shared-memory machine, or a software emulation of shared memory, or on a different communication architecture such as VIA.

The first two differ in the number of cache units that must be moved between the processors, while the latter differs from them in requiring fewer overall round trips. To explain the last statement, consider that in a cache-coherent shared memory system, a write to a memory location by one processor involves a round trip through the network, in addition to those required for invalidating any cached read copies. When this location is subsequently read by another processor (as happens in the radix application across rounds), another round trip is incurred. In contrast, in architectures such as VIA
that supports remote memory writes, both the write and the subsequent read can be accomplished simply by a single network transaction (assuming coherence can be ensured through other means). 

Estimate the data transfer costs in the radix application running on 4 processors for the following three cases. Express these costs in terms of the number and types (data versus invalidation) of network transactions and whether or not these involve round trips:

- An Origin-like hardware cache-coherent architecture with a cache block size of 128 bytes.
- A Beowulf-cluster like software distributed shared memory architecture with a cache block size of 4096 bytes.
- A VIA-like architecture with the ability to send a contiguous block of memory locations in local memory to a corresponding block of contiguous locations in remote memory using a single network transaction.

Probably the simplest way of assessing these costs is to profile the sequential program’s execution and record, for each round, the keys that would have been processed by a particular processor and how these keys would be distributed at the end of the round. This information can be used along with a simple model for how the keys are partitioned across the processors to estimate the number of network transactions in each of the above cases. For the first case above, you can assume that there are no capacity or conflict misses.

**Guidelines**

You should hand-in a writeup that contains (a) the code modifications for part 1; (b) the speedup measurements and a discussion of your optimizations for part 2; and (c) a description of your approach in part 3 along with an analysis of your estimates.

One final word of advice: START EARLY!
ANL Parallel Processing Macro Package Tutorial

Introduction

The Argonne National Laboratory's (ANL) parallel processing macro package provides a virtual machine that consists of a shared global memory and a number of processors with their own local memory. The macros themselves are a set of process control, synchronization, and communication primitives implemented as C-language m4 macros.

The use of macros has the advantage of portability. In fact, there are two layers of macros: a small set of primitive, machine-dependent macros written in terms of target machine functions and a larger set of machine-independent macros written in terms of the machine-dependent macros, so only few machine-dependent macros need to be ported to a new machine. Unfortunately, the use of macros also makes debugging more difficult, since error messages are given in reference to the C program generated from the original program.

This document describes a subset of the ANL macros as an introduction to their use.

Environment Specification Macros

Some of the macros assume the existence of certain data structures. The MAIN_ENV and the EXTERN_ENV macros contain the necessary definitions and declarations; MAIN_INITENV performs required initialization.

MAIN_ENV contains types and structures used internally in the macro package. It should appear in exactly one file (typically the main file) in the static definitions section before any other macro usage.

EXTERN_ENV contains definitions and external declarations and should appear in the static definitions section of each separately compiled module in which MAIN_ENV does not appear.

MAIN_INTENV is an executable macro that initializes data structures defined by MAIN_ENV. The code generated by this macro must be executed before that of any other macro, thus it typically appears very early in the program's main function.

MAIN_END must be the last thing in your main() routine. It is used for any cleanup necessary to support the programming environment.

Memory Allocation Macros

It is a good idea to declare a single structure, say gm_t, as global memory, and use a single call to G_MALLOC to allocate this structure, say in variable gm. Parts of global memory can then be referenced as, say, gm->someVar.
G_MALLOC(size) behaves like the Unix/C malloc call, except that the pointer returned points to globally shared memory which is accessible to all processes. For example,

```
    gm = (struct gm_t*) G_MALLOC(sizeof(struct gm_t));
```

where gm_t is a structure declared earlier.

G_FREE(ptr, size) de-allocates memory allocated by G_MALLOC, and is similar to the Unix/C free procedures.

P_MALLOC(size) behaves the same as the Unix/C malloc call, but is used for individual processes which returns a pointer that is not accessible to other processes.

**Process-Control Macros**

CREATE(entryProc) causes a process to be created and start executing the procedure entryProc. No arguments can be passed to the new process, or as parameters to entryProc. The process is a Unix-style process and, in fact, CREATE uses the fork system call. Note: At the point when a process is created, all of the parent's static data, including the pointer to global shared memory is copied once into a separate address space for the created process. The only memory that is shared is the memory explicitly allocated by G_MALLOC. Globally allocated data is static.

WAIT_FOR_END(nProcs) waits for nProcs processes created by this process to exit.

**Synchronization Macros**

There are macros provided for locking, barriers, and distributed loops. In each case, there is a macro for declaration (its name ends in DEC); the declaration macro should appear within a structure that is allocated with G_MALLOC, so that it will be globally shared and accessible to all processes. Another macro contains initialization code (its name ends in INIT); the initialization must occur before any use.

LOCKDEC(lockName) contains a lock declaration.

LOCKINIT(lockName) initializes the lock lockName.

LOCK(lockName) attempts to acquire ownership of the lock named lockName. If no other process currently owns the lock, then the process becomes the owner of the lock and proceeds. Otherwise, it is delayed until it can acquire the lock.

UNLOCK(lockName) relinquishes ownership of the lock given by lockName. If other processes are waiting to acquire the lock, one of them will succeed. Shen multiple locks need to be acquired, deadlocks can occur. Perhaps the simplest strategy to avoid deadlocks in this case is to have all processes acquire the locks in the same fixed order. If
the created processes all try to output to standard output at once, there can be trouble - so use a lock to access standard output, or let only the main process generate output.

BARDEC(barName) declares a barrier with the given name.

BARINIT(barName) is an executable macro that initializes the barrier.

BARRIER(barName, nProcs) stops all processes reaching this statement until nProcs processes have reached it. When that happens,
1. Barrier barName is reinitialized; it is not necessary to call BARINIT(barName) again.
2. All the processes continue on from the BARRIER statement.

**Distributed Loops: Get Subscript**

These macros aid in coordinating a distributed or self-scheduled loop. A self-scheduled loop is executed in parallel; each process dynamically acquires the next iteration to be executed (in this case, by first obtaining its corresponding index value).

GSDEC(name) declares an instance of a distributed loop.

GSINIT(name) initializes internal variables of the distributed loop.

GETSUB(name, subscript, maxSub, nProcs) sets subscript to the next available subscript. When all subscripts in the range 0 to maxSub (inclusive) have been returned, the following will happen to a process executing GETSUB, in this order:
1. The GETSUB operation is delayed until nProcs processes have requested an out-of-range subscript.
2. Loop instance name is reinitialized; it is not necessary to call GSINIT(name) again.
3. A value of –1 is returned for subscript.

**Timing Macros**

Execution time of part of whole programs can be measured using the CLOCK macro. It gives the current elapsed time in some time unit, not actual CPU time, which means that it is in general important that no other programs run during time measurements.

CLOCK(time) sets time to the current timer value, from 0 to $2^{32}$-1, where time is declared as

Unsigned int time;

The CLOCK macro will typically be used to computer time difference, so the fact that time may wrap around does not matter, as long as your program takes less than $2^{32}$ time units.