Prof. Dr. C. Stamm christoph.stamm@fhnw.ch Tel.: 056 202 78 32

Solution 1

1 Hypercube

a) The labels in a d-dimensional hypercube use d bits. Fixing any k of these bits, show that the nodes whose labels differ in the remaining $d - k$ bit positions form a $(d - k)$-dimensional sub-cube composed of 2^{d-k} nodes.

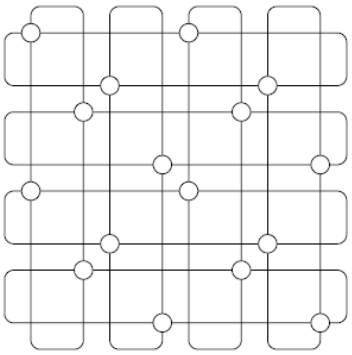
Consider a 2^d processor hypercube. By fixing k of the d bits in the processor label, we can change the remaining $d - k$ bits. There are 2^{d-k} distinct processors that have identical values at the remaining k bit positions.

A p-processor hypercube has the property that every processor has log p communication links, one each to a processor whose label differs in one bit position. To prove that the 2^{d-k} processors are connected in a hypercube topology, we need to prove that each processor in a group has $d - k$ communication links going to other processors in the same group.

Since the selected k bits are fixed for each processor in the group, no communication link corresponding to these bit positions exists between processors within a group. Furthermore, since all possible combinations of the $d - k$ bits are allowed for any processor, all $d - k$ processors that differ along any of these bit positions are also in the same group. Since the processor will be connected to each of these processors, each processor within a group is connected to $d - k$ other processors. Therefore, the processors in the group are connected in a hypercube topology.

b) One of the drawbacks of a hypercube-connected network is that different wires in the network are of different lengths (resulting in different data transfer times). It appears that 2D mesh networks with wraparound connections suffer from this drawback too. However, it is possible to fabricate a 2D wraparound mesh using wires of fixed length. Illustrate this layout by drawing such a 4 x 4 wraparound mesh.

The following figure illustrates a 4 x 4 wraparound mesh with equal wire lengths.



2 Summation with OpenMP

In this exercise we want to sum up all natural numbers between 1 and 10^7 within a parallel for loop. Of course, there is no need to do this in a real application, because we can always explicitly compute the result without any iteration. However, this simple example let us test different approaches in handling the synchronized access on shared variables:

```
long long sumPar1(const int n) {
   long long sum = 0;
   #pragma omp parallel for default(none) shared(sum)
   for (int i=1; i <= n; i++) {
      sum += i;
   }
   return sum;
}
```

a) List at least four possibilities that OpenMP provides for making the given source code correct.

sequential (2 ms)

reduction clause (1.4 ms)
critical section (1768 ms)
atomic update (168 ms)
explicit lock (2519 ms)

b) Program the four different variants in summation.cpp, measure the performance, and check the results by comparing them with the explicit computed result.

In OpenMP parallel access on shared variables is not automatically thread-safe. The best correct solution uses a private *sum* variable and makes use of the reduction clause, which reduces the synchronization overhead to a minimum:

```
long long sum = 0;
#pragma omp parallel for default(none) reduction(+: sum)
for (int i=1; i <= n; i++) {
   sum += i;
}
return sum;
```

The next three solutions use one of the three OMP synchronization constructs: a critical section, an atomic update, or an explicit lock:

```
long long sum = 0;
#pragma omp parallel for default(none) shared(sum)
for (int i=1; i <= n; i++) {
   #pragma omp critical
   sum += i;
}
return sum;
```

```
long long sum = 0;
#pragma omp parallel for default(none) shared(sum)
for (int i=1; i <= n; i++) {
   #pragma omp atomic
   sum += i;
}
return sum;
```

```
omp_lock_t myLock;
```

```
long long sum = 0;
omp_init_lock(&myLock);
#pragma omp parallel for default(none) shared(sum, myLock)
for (int i=1; i <= n; i++) {
   omp_set_lock(&myLock);
   sum += i;
   omp_unset_lock(&myLock);
}
omp_destroy_lock(&myLock);
return sum;
```

3 Image Processing

Parallelizing sub-optimal serial codes often has undesirable effects of unreliable speedups and misleading runtimes. For this reason, we advocate optimizing serial performance of codes before attempting parallelization.

"imageprocessing.cpp" contains in the subroutine `processSerial(…)` an image processing program for edge detection in RGB images. For handling images we use the portable and open source library *FreeImage* (http://freeimage.sourceforge.net/).

a) Compile (without any optimization flags), link and run the given program. Measure the runtime of this base variant (version 0). Since runtime measurements are always related to the performance of your test machine, write down the most important technical properties of your test machine.

Dell XPS 8500, Windows 10 Pro, 64-Bit
Intel Core i7-3770 CPU @ 3.4 GHz
12 GByte RAM

Version 0: 9743 ms (Debug version, no optimization)

b) Study the compiler optimizations of your compiler and compile the same program with the best compiler settings. Compute the speedup of this version 1 to version 0.

Version 1: 7800 ms (Release version, /O2 /Ob2 /Ot /GL), Speedup: 1.25

c) Now, have a closer look to the given program code. Copy the code in `processSerial(…)` to `processSerialOpt(…)` and try to optimize the new code as much as possible, but don't use parallelization. Use again the best suited compiler settings and compute the speedup of your optimized code compared to version 1.
Hints: The *FreeImage* methods `getPixelColor(…)` and `setPixelColor(…)` are very convenient but also very slow. Use instead `getScanLine(…)`, but try to minimize the number of calls of this subroutine. A speedup to version 1 of ca. 6 should be possible.

Version 2: 666 ms (Release version, /O2 /Ob2 /Ot /GL), Speedup: 11.70

d) The optimized serial implementation (version 1) should be a good starting point for parallelization. Try to parallelize the outer most for-loop. You can use OpenMP or the parallelized for-loop in C++11. The expected speedup to version 1 should be almost the number of CPU cores in your machine, because there is no need for synchronization.

OMP Version: 144 ms (Release version, /O2 /Ob2 /Ot /GL), Speedup: 4.66