Throughput Computing

From Documentation

Contents

[hide]

Overview

This tutorial provides information about how to utilize the SHARCNET throughput system, whale, to get fast results for problems that can be decomposed to run as independent concurrent serial jobs.

Prerequisites

Familiarity with C, bash and basic software development in a unix-like environment is useful to understand the presented examples, but the basic concepts should be accessible with knowledge of similar languages.

See Also

Managing a Large Number of Jobs, which presents a concise overview of methods to submit multiple jobs at once to the systems.

Throughput Background

perfectly / embarrassingly parallel

Throughput computing is a method for solving computational problems in a short amount of time. The method involves decomposing a problem into independent work units, which are then submitted as a large number of independent jobs to a computer system and run concurrently in parallel. Key to this approach is the requirement that the jobs to contribute to the overall solution in parallel, without having to communicate any values amongst themselves during the calculation or return the results in a particular order. This is often referred to as decomposing the workload into independent chunks, serial farming, embarrassingly parallel or perfectly parallel. This is commonly used in algorithms that are sampling a parameter space. A simple example follows of an integration program that can be decomposed in such a fashion along the integration path.

why NOT to use MPI or a threading model

It's very important to note that while one could use an MPI wrapper or implement threads to partition and allocate the chunks, it is not necessary for throughput computation and is actually less optimal than using a set of serial jobs. It's not

robust (if one process/thread fails they will all fail) and it will typically take far longer for the system to start the job since it needs to free up a lot of processors at once, rather than having the jobs "fill the gaps" when other serial jobs vacate a slot on a shared node. As such, using a lot of serial jobs on systems that give them priority will lead to the fastest turnaround (time to solution) and should always be used when possible instead of writing a threaded or MPI master/slave wrapper around the non-communicating program. In other words, it is only necessary to move to MPI or threads when there is a significant amount of communication amongst the parallel threads or processes, which occurs during a significant portion of the program's runtime.

That being said, one can use a throughput job submission model with a threaded or MPI program if the underlying program necessitates or is justified by their use.

Throughput Porting Process

The basic concept of throughput computing is to submit multiple independent instances of a program to compute a portion of the answer. This can be facilitated in a batch queuing job system by writing an auxiliary shell script or program that can be used to submit and collect the results. It may require pre or post-processing to address running the jobs concurrently. The following walk-through example illustrates the porting process and goes through the following steps:

- minor modifications to the original C program to compute a fraction of the total computation
- writing a bash shell script to submit the jobs
- writing a second bash shell script to accumulate the results after the jobs have completed

Not all of these steps are necessary and there are other ways of Managing a Large Number of Jobs, but the following example is a relatively straight-forward way to implement independent task parallelism in a portable way. There is no dependency on the underlying job management system or any other supporting libraries beyond access to the bash shell. One could conceive of a more elaborate way to incorporate all of the required logic in the C program itself but that is beyond the scope of this tutorial.

Example C Code: Integrating PI

We begin with this simple C code that integrates arctan from 0:1 to calculate PI:

```
#include <stdio.h>
#include <math.h>
int
main(int argc, char *argv[])
{

  double mypi,h,sum,x;
  int n,i;

  n=1000000;
  h=1.0/n;
  sum=0.0;

  printf("Calculating PI:\n");

  for (i = 1; i <= n; i+=1 ) {
    x = h * ( i - 0.5 ) ;    //calculate at center of interval
    sum += 4.0 / ( 1.0 + pow(x,2) ) ;
  }

  mypi = h * sum;

  printf("%f\n",mypi) ;

  return 0;
}
```

Running it gives:

```
[merz@wha780 integration]$ cc pi_orig.c
[merz@wha780 integration]$ ./a.out
Calculating PI:
```

```
3.141593
```

Modifying the code to run in parallel

In order to solve this problem in parallel, we will partition the for loop into discrete chunks that each serial job will calculate, and then after all of the jobs are finished we can simply sum the individual answers to get the final answer. In particular, if we used 10 chunks, then each chunk would compute 0.1 of the integration path, ie, chunk one computes (0:0.1], chunk two computes (0.1:0.2], etc.

The modified C code, which reads the particular ***chunk to calculate*** and ***total number of chunks*** on the command line as standard input, follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int
main(int argc, char *argv[])
{

  double mypi,h,sum,x;
  int n,i,chunk,tot_chunks,chunk_size,n_low,n_high;

  n=1000000;
  h=1.0/n;
  sum=0.0;

// check to make sure we've been given two arguments

  if (argc != 3) {
    printf("Usage:  pi.x chunk tot_chunks\n");
    exit(0);
  }

// break up number of total intervals based on the
// number of chunks and assign loop indexes for this
// instance of the program to n_low and n_high

  chunk= atoi(argv[1]);
  tot_chunks= atoi(argv[2]);
  chunk_size=n/tot_chunks;
  n_low=(chunk-1)*chunk_size+1;
  n_high=chunk*chunk_size;
  if (chunk == tot_chunks) {
    n_high=n;
  }
  printf("Calculating PI for interval %d of %d from %d to %d \n", chunk,tot_chunks,n_low,n_high);

// modify the loop to use n_low and n_high

  for (i = n_low; i <= n_high; i+=1 ) {
    x = h * ( i - 0.5 ) ;    //calculate at center of interval
    sum += 4.0 / ( 1.0 + pow(x,2) ) ;
  }

  mypi = h * sum;

  printf("%f\n",mypi) ;

  return 0;
}
```

There are smarter ways to decompose this workload (ie. spreading out the remainder instead of allocating it all to the final chunk), but this should suffice to get started.

bash Job Submission Script

This script submits a series of jobs to the batch queue system via sqsub. It specifies 2 additional arguments to the executable: the current iteration, and the total number of iterations. The iterations are indexed from 1:N

It is expected that the executable would use these input arguments to decide which portion of the total workload it accounts for. The logic associated with partitioning the workload should be addressed in the executable program.

The following variables should be set appropriately before calling this script:

- DEST_DIR
  - path to the base directory for submission of the job
  - it should exist and contain the executable
- EXENAME
  - name of the executable
- NUM_ITERS
  - number of jobs to submit
  - eg. integration intervals
- RUNTIME
  - how long to let the job run for, in sqsub format
- OUTFILE
  - optionally change the format to something more meaningful

The script follows - it should be put in a file (I've used ***sub_script*** below) and made executable by issuing the command **chmod +x sub_script**:

```
#!/bin/bash
DEST_DIR=/work/merz/integration
EXENAME=pi.x
NUM_ITERS=20
RUNTIME=10m
echo "${DEST_DIR}/${EXENAME}"
if [ -e "${DEST_DIR}/${EXENAME}" ]
then
  cd ${DEST_DIR}
  for z_interval in `seq 1 ${NUM_ITERS}`; do
    echo "Submitting interval: ${z_interval} of ${NUM_ITERS}"
    OUTFILE="OUTPUT-${z_interval}.txt"
    sqsub -r ${RUNTIME} -q serial -o ${OUTFILE} ./${EXENAME} ${z_interval} ${NUM_ITERS}
 done;
else
  echo "couldn't find the above executable, exiting"
fi
```

For brevity of the output, I've reduced NUM_ITERS to 4 and run the script with the modified program to illustrate. In practice, users will want to use a lot of chunks, in the tens to hundreds:

```
[merz@wha780 integration]$ pwd; ls; cc pi.c -o pi.x
/work/merz/integration
accumulate  pi.c  pi_orig.c  sub_script
[merz@wha780 integration]$ ./sub_script
/work/merz/integration/pi.x
Submitting interval: 1 of 4
THANK YOU for providing a runtime estimate of 10m!
submitted as jobid 4109486
Submitting interval: 2 of 4
THANK YOU for providing a runtime estimate of 10m!
submitted as jobid 4109487
Submitting interval: 3 of 4
THANK YOU for providing a runtime estimate of 10m!
submitted as jobid 4109488
Submitting interval: 4 of 4
THANK YOU for providing a runtime estimate of 10m!
submitted as jobid 4109489
<wait 30 seconds...>
[merz@wha780 integration]$ sqjobs
[merz@wha780 integration]$ ls
accumulate  OUTPUT-1.txt  OUTPUT-2.txt  OUTPUT-3.txt  OUTPUT-4.txt  pi.c  pi_orig.c  pi.x  sub_script
```

Each of the OUTPUT* files will contain a portion of the answer, eg.

```
[merz@wha780 integration]$ cat OUTPUT-1.txt
```

```
Calculating PI for interval 1 of 4 from 1 to 250000
0.979915

------------------------------------------------------------
Sender: LSF System <lsfadmin@wha482>
<snip>
```

So we need to write a post-processing utility that can quickly sum up the answer if we're going to use a lot of chunks.

bash Post-Processing Script

In this case we just need a program or script to read in the second line of each of the OUTPUT files, and then sum the results. In this case it's pretty straight-forward and doesn't take a long time to run, so instead of writing a full program we'll just use a bash script. The following script will calculate the answer (note the use of the **bc** command - there are no floating point numbers in bash):

```
#!/bin/bash
total=0;
value=0;
for file in OUTPUT*; do
  value=`head -2 $file | tail -1`;
  total=`echo $total + $value | bc`;
done
echo $total;
```

Copying this into a file (I've chosen ***accumulate***), setting it executable with **chmod +x accumulate** and then running it in the directory where I submitted the job returns the correct answer:

```
[merz@wha780 integration]$ ./accumulate
3.141593
```

Retrieved from "https://www.sharcnet.ca/help/index.php/Throughput_Computing"
Category: Tutorials