Cornell University
Center for Advanced Computing

Lab: OpenMP

Steve Lantz
Senior Research Associate
Cornell CAC

Workshop: Introduction to Parallel Computing on Ranger, May 29, 2009

Based on materials developed at TACC

OpenMP lab: getting started

- Log on to Ranger using your train## account:

```
ssh -X train##@ranger.tacc.utexas.edu
```

Your assigned number

- Untar the file lab_openmp.tar file (in ~train00) into your directory

```
tar –xvf ~train00/lab_openmp.tar
```

- cd down into the lab_openmp directory

```
cd lab_openmp
```

- On the following pages, plots of library performance are only shown for the IBM ESSL library; similar plots can be obtained for Intel MKL

Using OpenMP on Ranger

- Use front end for experimenting with short runs
- Submit the job after making executables
- Proceed with the rest of the lab while the batch job is running

Execute these commands

```
module unload mvapich2
module swap pgi intel
module load mvapich2
module load mkl
make work
make saxpy
make saxpy2
qsub job
```

OpenMP "Hello, World"

- Look at the code in the source files hello.c and hello.f: they simply report thread IDs in a parallel region
- Compile hello.c and hello.f for multitasking and execute with 1 to 5 threads (see also do_c_hello and do_f90_hello scripts)

F90 program: compile and run

```
make hello_f90
setenv OMP_NUM_THREADS 3
./hello_f90
```

C program: compile and run

```
make hello_c
setenv OMP_NUM_THREADS 3
./hello_c
```

- Do **"`make run_hello_c`"** or **"`make run_hello_f90`"** for automated execution of 1 to 16 threads

Parallel Region Example

- Look at the code in file work.f90
  - Threads perform some work in a subroutine called pwork
  - The timer returns wall-clock time
- Compile work.f for multitasking and execute with 3 and 4 threads
  - There can be a wide variation in the runs when the system is busy
  - Use the **`top`** command see the load on the system
- To compile and run the work program:

```
make work
setenv OMP_NUM_THREADS 3
./work
```

- Check the system load

  `top` {hit the “1” key to see the CPU loads; hit the “q” key to quit }

Parallel efficiency

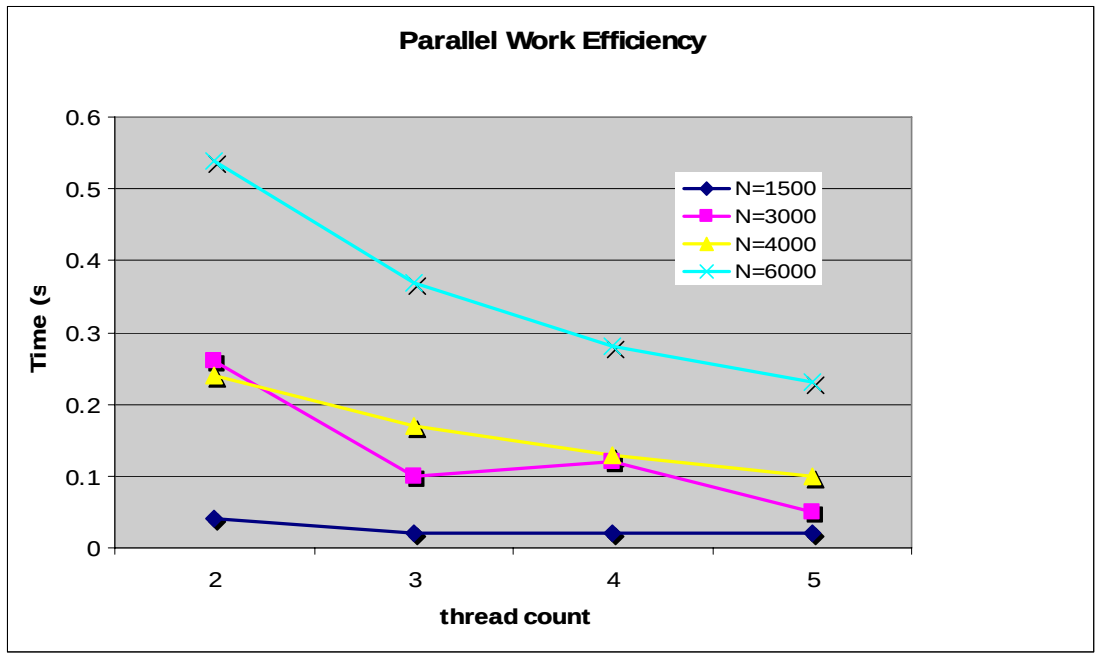
- Change N in the code and execute **"make run_work"** again.
- Why does the parallel efficiency improve with data size *N*?
- The graph below shows the times on a dedicated (non-busy) system.

Execute:

```
make run_work
```

…to run the executable for 1 through 5 threads

(Example was run on Champion, IBM system)



SAXPY Scheduling

- Look at the code in file saxpy.f
  - The nested loop performs a simple SAXPY type operation
  - N determines the size of the problem (N=1024*40 is the default)
- Compare the performance under different types of scheduling
  1. Compile the code
  2. Set the scheduling type: “static”, “dynamic”, or “dynamic, 64”
  3. Run the program with 1-5 threads, repeat for different scheduling
  4. Repeat steps 1-3 for different values of N

Compile and run the saxpy program:

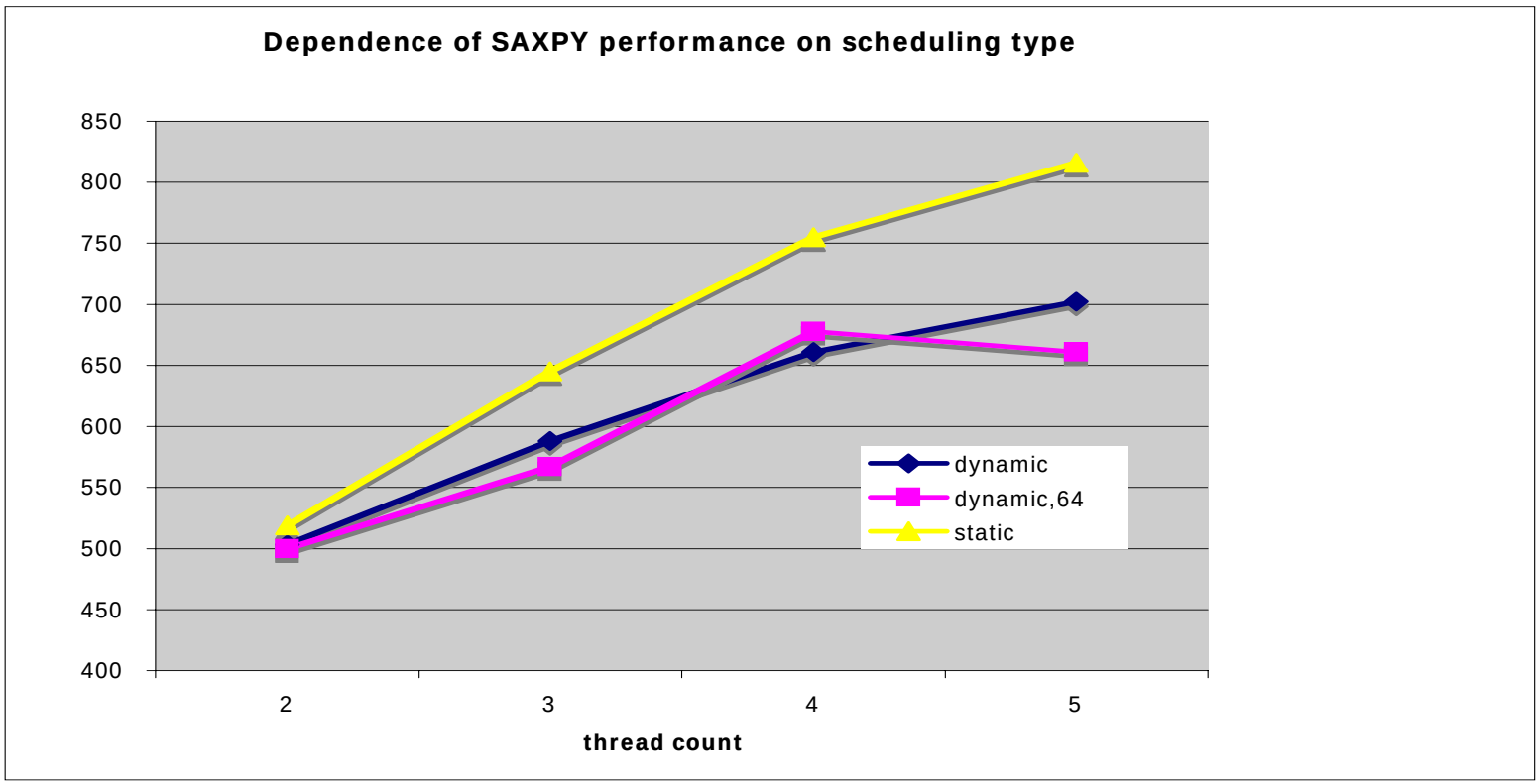
```
make saxpy
setenv OMP_SCHEDULE static
make run_saxpy
```

Look at the system load:

top {hit the “q” key to quit}

SAXPY performance

(Example taken from Champion w/ ESSL - make sure you see the MKL library results in the next section, SAXPY loops are notoriously slow on RISC architectures)



Intel MKL DAXPY (library routine)

- Look at the code in file saxpy2.f
  - The nested loop performs a DAXPY operations for each outer loop
  - The DAXPY routine comes from the MKL Library, which must be loaded
- Execute the do_saxpy2 script and compare the result from your previous saxpy runs
  - Change the parameter N and the scheduling clause to agree with your previous runs
- Use the **`top`** command to watch the load on the system

```
make saxpy2
make run_saxpy2
```

Library vs. hand coding

(Example taken from Champion w/ ESSL)