

# Performance Tools

Tulin Kaman

Department of Applied Mathematics and Statistics

Stony Brook/BNL New York Center for Computational Science

[tkaman@ams.sunysb.edu](mailto:tkaman@ams.sunysb.edu)

Aug 24, 2012

# Performance Tools

- Community Tools:

- GNU Profiler: tool provided with the GNU compiler



- Tuning and Analysis Utilities (TAU)

- PAPI (Performance Application Programming Interface)

- High Performance Computing Toolkit (HPCT) for IBM Blue Gene

- Message Passing Interface (MPI) Profiler and Tracer tool

- Xprofiler for CPU profiling

- Hardware Performance Monitoring (HPM) library

- Modular I/O (MIO) library

# Tuning and Analysis Utilities - TAU

- Performance evaluation tool
- Profiling and tracing toolkit for performance analysis of parallel programs written in C, C++, Fortran, Java and Python
- Support for multiple parallel programming paradigms: MPI, Multi-threading, Hybrid (MPI + Threads)
- Access to hardware counters using PAPI
- **Memory Profiling**

# Memory events TAU can evaluate :

- How much heap memory is currently used ?

Memory utilization options

- How much can a program grow before it runs out of free memory on the heap ?

Memory headroom evaluation options

- Memory leaks in C/C++ programs TAU will track malloc through the execution issuing user event when the program fails to the allocated memory.

# Dynamic instrumentation through library preloading

- Enabled by command-line options to **tau\_exec**  
**tau\_exec** [options] [--] <exe> <exe options>

Options:

- -io track I/O
- -memory track memory
- -cuda track GPU events via CUDA
- -opencl track GPU events via OpenCL

```
$ tau_exec -memory ./a.out
```

```
$ mpirun -np 4 tau_exec -memory ./a.out
```

# Memory Utilization Option

## TAU\_TRACK\_MEMORY()

- Call in your code
- When it is called, an interrupt is generated every 10 seconds (DEFAULT)

TAU\_SET\_INTERRUPT\_INTERVAL(value) : changes the interrupt interval

- explicitly enabled or disabled by calling  
TAU\_ENABLE\_TRACKING\_MEMORY()  
TAU\_DISABLE\_TRACKING\_MEMORY()

# Memory Utilization Option

## TAU\_TRACK\_MEMORY\_HERE()

- Triggers memory tracking at a given execution point

**Example:** /gpfs/home1/tulin/TAUP/tau-2.18/examples/memory/simple.cpp

```
int main(int argc, char **argv)
{
    TAU_PROFILE("main()", " ", TAU_DEFAULT);
    TAU_PROFILE_SET_NODE(0);

    TAU_TRACK_MEMORY_HERE();

    int *x = new int[5*1024*1024];
    TAU_TRACK_MEMORY_HERE();
    return 0;
}
```

USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0

NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.	Event Name
2	2.049E+04	2.891	1.024E+04	1.024E+04	Memory Utilization (heap, in KB)

# Memory Utilization Option

- -PROFILEMEMORY

- Track heap memory utilization at each function entry
- Each configuration creates a unique Makefile

```
./configure -arch=bgp -pdt=/gpfs/home1/tulin/TAUP/pdtoolkit-3.14  
-pdt_c++=xlc -mpi -PROFILEMEMORY
```

- Set environment variables

```
export PATH=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/bin:$PATH
```

- Use TAU\_MAKEFILE

```
export TAU_MAKEFILE=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/lib/Makefile.tau-memory-mpi-pdt
```

- Compile your code using TAU compiler scripts: tau\_cc.sh tau\_cxx.sh tau\_f90.sh
- Run to generate profile files.

# Memory Headroom Option

- examines how much a program can grow before it runs out of free memory on the heap

## TAU\_TRACK\_MEMORY\_HEADROOM()

- Call in your code
- vary the size of the callstack by setting the environment variable TAU\_CALLSTACK\_DEPTH (default is 2)
- When it is called, an interrupt is generated every 10 seconds (DEFAULT)

TAU\_SET\_INTERRUPT\_INTERVAL(value): changes the interrupt interval

- explicitly enabled or disabled by calling  
TAU\_ENABLE\_TRACKING\_MEMORY\_HEADROOM()  
TAU\_DISABLE\_TRACKING\_MEMORY\_HEADROOM()

# Memory Headroom Option

## TAU\_TRACK\_MEMORY\_HEADROOM\_HERE()

- evaluates how much memory it can allocate and associates it with the callstack
- **Example:** /gpfs/home1/tulin/TAUP/tau-2.18/examples/headroom/here

```
int f1(void)
{
    double *ary;
    TAU_PROFILE("f1()", "(sleeps 1 sec, calls f2, f4)", TAU_USER);
    printf("Inside f1: sleeps 1 sec, calls f2, f4\n");
    ary = new double [1024*1024*50];
    TAU_TRACK_MEMORY_HEADROOM_HERE();
    sleep(1);
    f2();
    f4();
    return 0;
}

int main(int argc, char **argv)
{
    TAU_PROFILE("main()", "(calls f1, f5)", TAU_DEFAULT);
    TAU_PROFILE_SET_NODE(0);
    printf("Inside main: calls f1, f5\n");

    TAU_TRACK_MEMORY_HEADROOM();
    f1();
    f5();
}
```

# Memory Hardware Option

- -PROFILEHEADROOM

- Track free memory (or headroom) at each func entry.
- Each configuration creates a unique Makefile

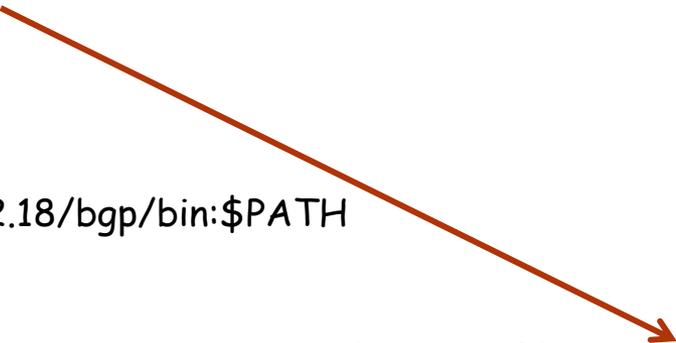
```
./configure -arch=bgp -pdt=/gpfs/home1/tulin/TAUP/pdtoolkit-3.14  
-pdt_c++=xlc -mpi -PROFILEHEADROOM
```

- Set environment variables

```
export PATH=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/bin:$PATH
```

- Use TAU\_MAKEFILE

```
export TAU_MAKEFILE=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/lib/Makefile.tau-headroom-mpi-  
pdt
```



- Compile your code using TAU compiler scripts: tau\_cc.sh tau\_cxx.sh tau\_f90.sh
- Run to generate profile files.

# How to detect memory leaks?

```
tulin@fenp:~/TAUP/tau-2.18/examples/memoryleakdetect/c> cat simple.c
```

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

```
#include <malloc.h>
```

```
/* there is a memory leak in bar when it is invoked
```

```
int bar(int value)
```

```
{
    printf("Inside bar: %d\n", value);
    int *x;

    if (value > 5)
    {
        printf("looks like it came here from g!\n");
        x = (int *) malloc(sizeof(int) * value);
        x[2]= 2;
        /* do not free it! create a memory leak, unless
        if (value > 15) free(x);
    }
    else
    { /* value <=5 no leak */
        printf("looks like it came here from foo!\n");
        x = (int *) malloc(sizeof(int) * 45);
        x[23]= 2;
        free(x);
    }
    return 0;
}
```

5 < value <= 15

```
int g(int value)
```

```
{
    printf("Inside g: %d\n", value);
    return bar(value);
}
```

```
int foo(int value)
```

```
{
    printf("Inside f: %d\n", value);

    if (value > 5) g(value);
    else bar(value);

    return 0;
}
```

```
int main(int argc, char **argv)
```

```
{
    int *x;
    int *y;
    printf ("Inside main\n");

    foo(12);
    foo(20);
    foo(2);
    foo(13);
}
```

```
tulin@fenp:~/TAUP/tau-2.18/examples/memoryleakdetect/c> ls
Makefile simple.c
tulin@fenp:~/TAUP/tau-2.18/examples/memoryleakdetect/c> make
```

```
Debug: Parsing with PDT Parser
Executing> /gpfs/home1/tulin/TAUP/pdtoolkit-3.14/bgp/bin/cparse s
T_H_LESS_HEADERS -DTAU_XLC -DTAU_MPI -DTAU_MPI_THREADED -DTAU_PRO
GNORE_CXX_SEEK -DTAU_BGP -I/bgsys/drivers/ppcfloor/arch/include/c
vers/ppcfloor/comm/include
```

```
Debug: Instrumenting with TAU
Executing> /gpfs/home1/tulin/TAUP/tau-2.18/bgp/bin/tau_instrument
```

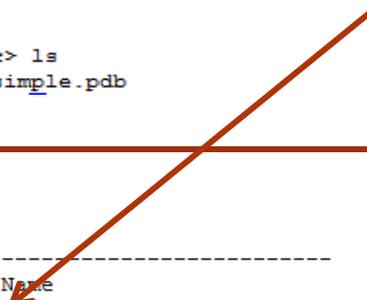
```
Debug: Compiling with Instrumented Code
Executing> mpixlc_r -I. -c simple.inst.c -DPROFILING_ON -DTAU_STD
Y -DTAU_MPIATTRFUNCTION -DTAU_MPITYPEEK -DTAU_MPIADDERORROR -DTAU_L
bgsys/drivers/ppcfloor/arch/include -I/bgsys/drivers/ppcfloor/arc
fs/home1/tulin/TAUP/tau-2.18/include/Memory -o simple.o
Looking for file: simple.o
```

```
Debug: Linking with TAU Options
Executing> mpixlc_r simple.o -o simple -lfmpich.cnk -L/gpfs/home1
-ldcmf.cnk -lphthread -L/bgsys/drivers/ppcfloor/runtime/SPI -lSPI
libmc++ -lstdc++
```

```
tulin@fenp:~/TAUP/tau-2.18/examples/memoryleakdetect/c> ls
Makefile simple simple.inst.c simple.o simple.pdb
```

```
Inside main
Inside f: 12
Inside g: 12
Inside bar: 12
looks like it came here from g!
Inside f: 20
Inside g: 20
Inside bar: 20
looks like it came here from g!
Inside f: 2
Inside bar: 2
looks like it came here from foo!
Inside f: 13
Inside g: 13
Inside bar: 13
looks like it came here from g!
```

## two Memory leaks



USER EVENTS Profile :NODE 0, CONTEXT 0, THREAD 0

NumSamples	MaxValue	MinValue	MeanValue	Std. Dev.	Event Name
2	52	48	50	2	MEMORY LEAK! malloc size <file=simple.inst.cpp, line=18> : int g(int) => int bar(int)
1	80	80	80	0	free size <file=simple.inst.cpp, line=21>
1	80	80	80	0	free size <file=simple.inst.cpp, line=21> : int g(int) => int bar(int)
1	180	180	180	0	free size <file=simple.inst.cpp, line=28>
1	180	180	180	0	free size <file=simple.inst.cpp, line=28> : int foo(int) => int bar(int)
3	80	48	60	14.24	malloc size <file=simple.inst.cpp, line=18>
3	80	48	60	14.24	malloc size <file=simple.inst.cpp, line=18> : int g(int) => int bar(int)
1	180	180	180	0	malloc size <file=simple.inst.cpp, line=26>
1	180	180	180	0	malloc size <file=simple.inst.cpp, line=26> : int foo(int) => int bar(int)

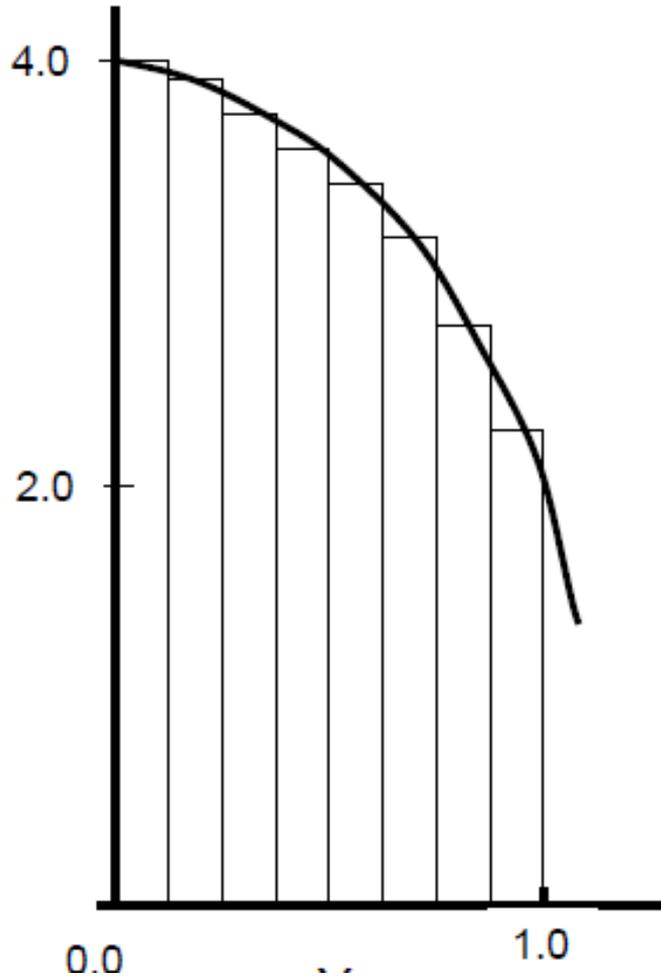
# Shared Memory Programming: OpenMP

## **OpenMP = Open Multi-Processing**

- The OpenMP Application Program Interface (API) for writing shared memory parallel programs.
- Supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures.
- Consists of
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- OpenMP program is portable. Compilers have OpenMP support.
- Requires little programming effort.

# Example: PI

3.14159265358979323846264338327950288419716



$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

NUMERICAL INTEGRATION

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

# Example: PI

```
#include <stdio.h>
#include <omp.h>

#define NUMSTEPS 1000000000
#define NUMTHRDS 4

int main ()
{
    int i, nthreads, tid;
    double x, step, Pi=0.0, sum=0.0;
    step = 1.0/NUMSTEPS;

    omp_set_num_threads(NUMTHRDS);

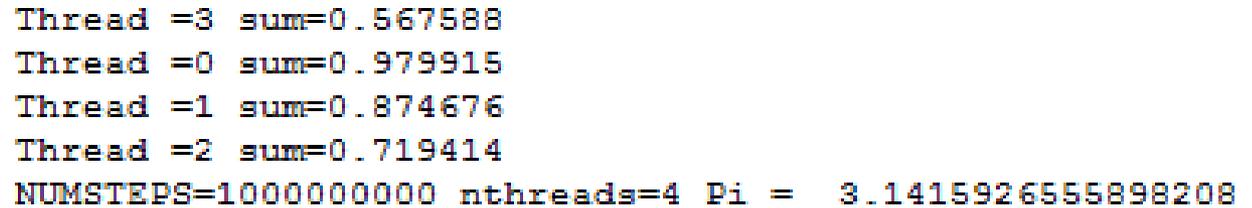
#pragma omp parallel private(x,tid) firstprivate(sum)
{
    tid = omp_get_thread_num();
    if (tid == 0)
        nthreads = omp_get_num_threads();

#pragma omp for
    for(i = 0; i <= NUMSTEPS; ++i)
    {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }
    printf("Thread =%d sum=%f\n",tid, sum*step);

#pragma omp critical
    Pi += step * sum;
}
printf ("NUMSTEPS=%d nthreads=%d Pi = %.16f \n", NUMSTEPS, nthreads, Pi);

return 0;
}
```

```
Thread =3 sum=0.567588
Thread =0 sum=0.979915
Thread =1 sum=0.874676
Thread =2 sum=0.719414
NUMSTEPS=1000000000 nthreads=4 Pi = 3.1415926555898208
```



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

```
#include <omp.h>
```

```
#define NUMSTEPS 1000000000
```

```
int main ()
```

```
{  
    int i, nthreads, tid;  
    double x, step, Pi=0.0, sum=0.0;  
    step = 1.0/NUMSTEPS;
```

```
#pragma omp parallel private(x,tid) firstprivate(sum)
```

```
{  
    tid = omp_get_thread_num();  
    if (tid == 0)  
        nthreads = omp_get_num_threads();  
  
    #pragma omp for  
        for(i = 0; i <= NUMSTEPS; ++i)  
    {  
        x = (i+0.5)*step;  
        sum += 4.0 / (1.0+x*x);  
    }  
    printf("Thread =%d sum=%f\n",tid, sum*step);
```

```
#pragma omp critical  
    Pi += step * sum;
```

```
}
```

```
printf ("NUMSTEPS=%d nthreads=%d Pi = %.16f \n", NUMSTEPS, nthreads, Pi);
```

```
return 0;
```

```
}
```

```
Thread =3 sum=0.567588
```

```
Thread =0 sum=0.979915
```

```
Thread =1 sum=0.874676
```

```
Thread =2 sum=0.719414
```

```
NUMSTEPS=1000000000 nthreads=4 Pi = 3.1415926555898208
```



Set environment variable

OMP\_NUM\_THREADS=4

# OMP\_GET\_WTIME()

- portable wall clock timing routine
- returns a double-precision floating point value equal to the number of elapsed seconds

## Fortran

DOUBLE PRECISION FUNCTION  
OMP\_GET\_WTIME()

## C/ C++

```
#include <omp.h> double  
omp_get_wtime(void)
```

```

#include <stdio.h>
#include <omp.h>

#define NUMSTEPS 1000000000

int main ()
{
    int i, nthreads, tid;
    double x, step, Pi1=0.0, sum=0.0;
    double start,end;
    step = 1.0/NUMSTEPS;

    start = omp_get_wtime();

#pragma omp parallel private(x,tid) \
    reduction(+:sum)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
            nthreads = omp_get_num_threads();

        #pragma omp for
        for(i = 0; i <= NUMSTEPS; ++i)
        {
            x = (i+0.5)*step;
            sum += 4.0 / (1.0+x*x);
        }

        end = omp_get_wtime();
        printf("Thread =%d omp_time=%f sec\n",nthreads, (double)(end-start));

        Pi1 += step * sum;
        printf ("NUMSTEPS=%d nthreads=%d Pi1 = %.16f \n", NUMSTEPS, nthreads, Pi1);

        return 0;
    }
}

```

Thread =1 omp\_time=41.176492 sec  
 Thread =2 omp\_time=20.588791 sec  
 Thread =4 omp\_time=10.295631 sec

```

int main ()
{
    int i, nthreads, tid;
    double x, step, Pi1=0.0, Pi2=0.0, sum=0.0, sum_single=0.0;
    double start,end;
    step = 1.0/NUMSTEPS;

    start = omp_get_wtime();

#pragma omp parallel private(x,tid) \
    reduction(+:sum)
{
    tid = omp_get_thread_num();
    if (tid == 0)
        nthreads = omp_get_num_threads();

#pragma omp for
    for(i = 0; i <= NUMSTEPS; ++i)
    {
        x = (i+0.5)*step;
        sum += 4.0 / (1.0+x*x);
    }

#pragma omp single
    for(i = 0; i <= NUMSTEPS; ++i)
    {
        x = (i+0.5)*step;
        sum_single += 4.0 / (1.0+x*x);
    }
}

    end = omp_get_wtime();
    printf("Thread =%d omp_time=%f sec\n",nthreads, (double)(end-start));

    Pi1 += step * sum;
    printf ("NUMSTEPS=%d nthreads=%d Pi1 = %.16f \n", NUMSTEPS, nthreads, Pi1);

    Pi2 = 0.0;
    Pi2 += step * sum_single;
    printf ("NUMSTEPS=%d nthreads=%d Pi2 = %.16f \n", NUMSTEPS, nthreads, Pi2);

    return 0;
}

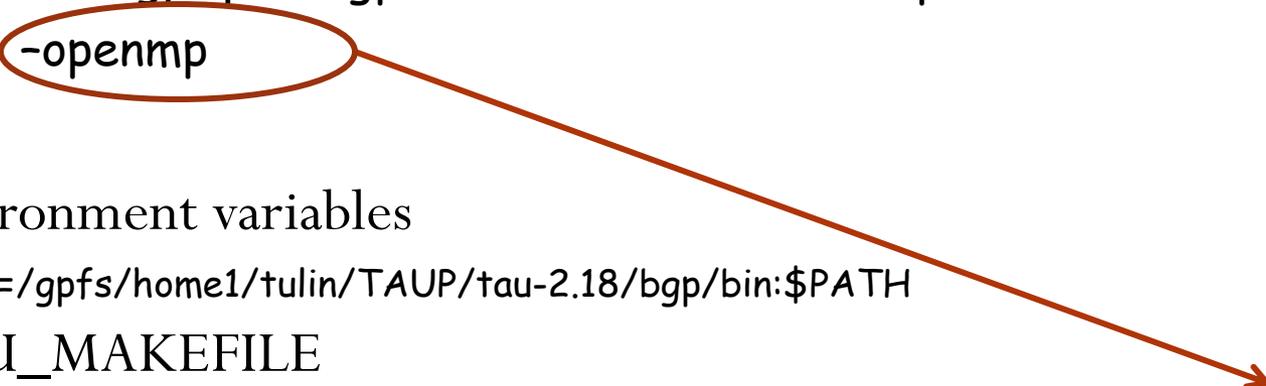
```

Thread =1 omp\_time=82.352963 sec  
 Thread =2 omp\_time=61.765259 sec  
 Thread =4 omp\_time=51.472104 sec

# OpenMP with TAU

- Each configuration creates a unique Makefile

```
./configure -arch=bgp -pdt=/gpfs/home1/tulin/TAUP/pdtoolkit-3.14  
-pdt_c++=xlc -openmp
```



- Set environment variables

```
export PATH=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/bin:$PATH
```

- Use TAU\_MAKEFILE

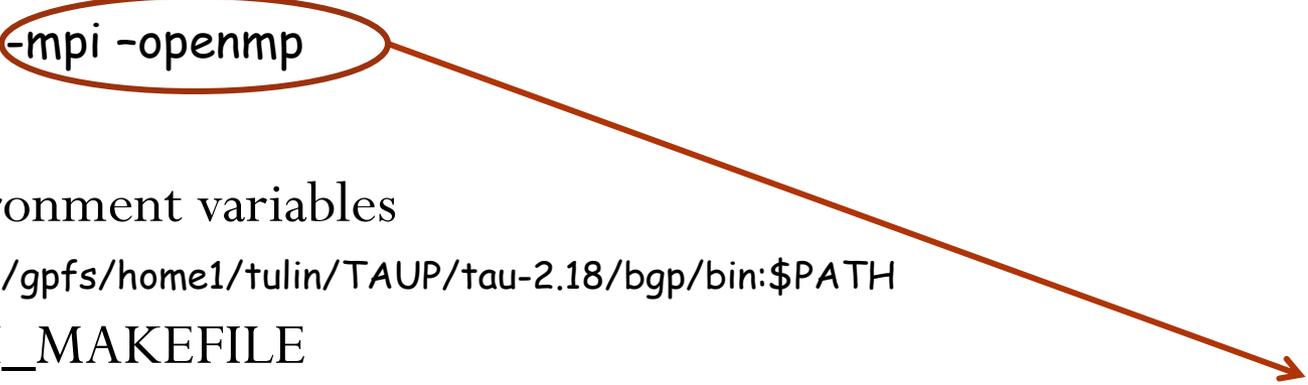
```
export TAU_MAKEFILE=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/lib/ Makefile.tau-pdt-openmp
```

- Compile your code using TAU compiler scripts: tau\_cc.sh tau\_cxx.sh  
tau\_f90.sh
- Run to generate profile files.

# OpenMP + MPI with TAU

- Each configuration creates a unique Makefile

```
./configure -arch=bgp -pdt=/gpfs/home1/tulin/TAUP/pdtoolkit-3.14  
-pdt_c++=xlc -mpi -openmp
```



- Set environment variables

```
export PATH=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/bin:$PATH
```

- Use TAU\_MAKEFILE

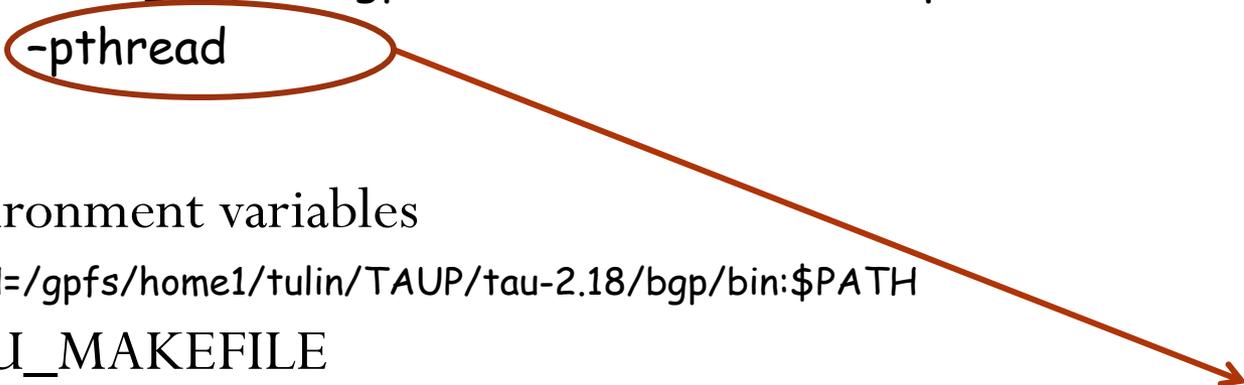
```
export TAU_MAKEFILE=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/lib/ Makefile.tau-mpi-pdt-openmp
```

- Compile your code using TAU compiler scripts: tau\_cc.sh tau\_cxx.sh tau\_f90.sh
- Run to generate profile files.

# Pthread with TAU

- Each configuration creates a unique Makefile

```
./configure -arch=bgp -pdt=/gpfs/home1/tulin/TAUP/pdtoolkit-3.14  
-pdt_c++=xlc -pthread
```



- Set environment variables

```
export PATH=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/bin:$PATH
```

- Use TAU\_MAKEFILE

```
export TAU_MAKEFILE=/gpfs/home1/tulin/TAUP/tau-2.18/bgp/lib/Makefile.tau-pthread-pdt
```

- Compile your code using TAU compiler scripts: tau\_cc.sh tau\_cxx.sh  
tau\_f90.sh
- Run to generate profile files.

# Stony Brook Center for Computational Science

Tutorial video and presentations are

<http://www.stonybrook.edu/sbccs/tutorials.shtml>

Tuning and Analysis Utilities: TAU

<http://www.cs.uoregon.edu/Research/tau/home.php>