Multi-Layer Thread Package

A Many-to-Many Thread Platform for SMP Linux

Michael Dipperstein

University of California, Santa Barbara

mdipper@cs.ucsb.edu

Overview

- What is Multi-Layer Thread Package (MLTP)
  - User-Level Threads
  - Kernel-Level Threads
  - Related Work and Comparison
  - MLTP Solution
- MLTP Architecture
  - Process Layer
  - User Layer

Overview

- Implementation of Layering
- Thread Scheduling
- User-Level Primitives
  - Thread Creation
  - Thread Yielding and Termination
  - Critical Section Locking
  - Barrier Synchronization
  - Conditional Waiting and Signaling

Overview

- Performance Results
  - Context Switching
  - Pi Approximation
  - Matrix Multiplication
  - SPLASH-2 Ocean
- Contribution/Conclusion

What is Multi-Layer Thread Package (MLTP)

- Multi-Layer Thread Package (MLTP) is a library implementing many-to-many (M-to-N) threads for SMP Linux
- MLTP contains an efficient and robust set of primitives for thread based computations
  - Exploits SMP architecture
  - Micro benchmarks and application based performance studies demonstrate efficiency and robustness
- MLTP is open-source and easily extensible

User-Level Threads

- Existing SMP Linux thread packages consist of either many-to-one or one-to-one architectures
- User-Level Thread Packages (Many-to-One)
  - Many user-level threads are executed in the context of a single process
  - Context switching is quick
  - Easy to implement
  - Single thread may block entire process
  - Cannot take advantage of multiple processors

Kernel-Level Threads

- Kernel-Level Thread Packages (One-to-One)
  - Each thread is executed in the context of a process
  - Scheduler is aware of each thread
  - Threads may be run on multiple processors
  - Context switching between threads is expensive
  - Overhead for each thread is large
  - Number of threads limited by the number of processes that the Kernel can run
    - Default of 512 may be recompiled for up to 1024

Related Work and Comparison

- UW Scheduler Activactions
  - Kernel up-calls give some scheduling control to User-Level threads
  - Requires kernel modifications
- Sun Threads
  - User-Level threads run on top of a pool of Lightweight Processes (LWP)
  - May have less than optimal performance if machine not coded for machine architecture

Related Work and Comparison

- UIUC Nano Threads
  - Exports resource allocation and processor state to User-Level threads
  - Imports user-level thread scheduling into the Kernel
  - Kernel modifications required
- UCSB TMPI Thread Package
  - Two layer cooperative thread package limiting process
  - Package handles architecture specific requirements
  - Kernel-Level threads distributed among programs

MLTP Solution

- MLTP provides a platform to gain performance advantages found in some Non-Linux thread packages
- MLTP provides a solution similar to Sun's thread implementation
  - User-Level threads execute in the context of Linux processes sharing resource allocations
- May be extended to add optimizations of other thread packages

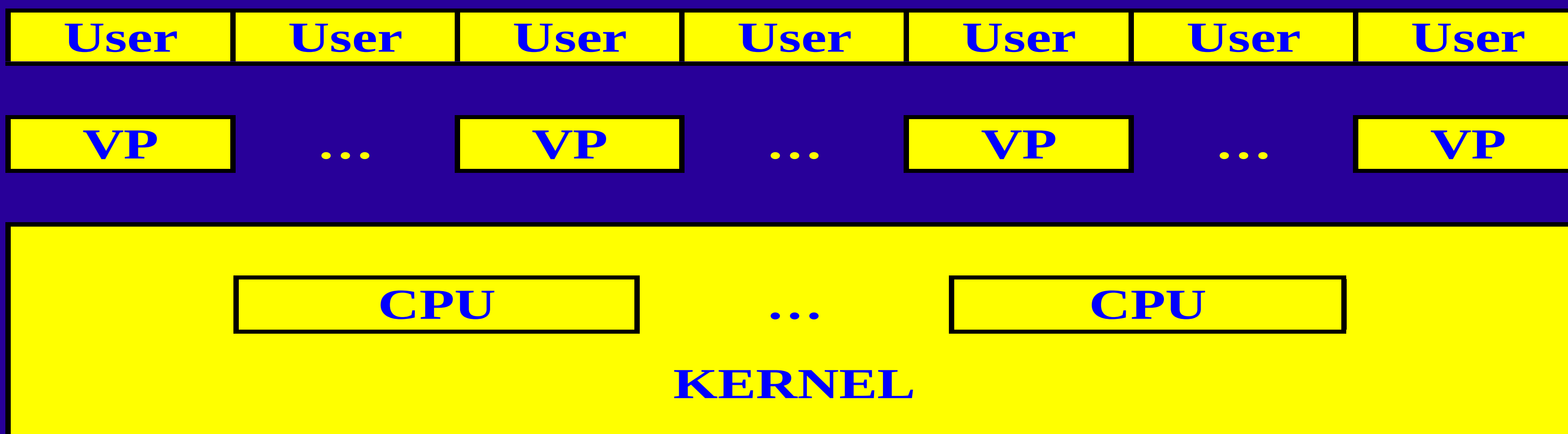
Why MLTP?

- Multi-Layer Thread Package (Many-to-Many)
  - Most of the advantages of both thread architectures
  - Scheduler is aware of process layer
  - Threads may run on multiple processors
  - Context switching between threads is inexpensive
  - Largest thread overhead is stack size
  - Number of threads limited by resources available for each thread's stack
    - Maximum addressable space on a Pentium is 2GB

MLTP Architecture

- MLTP two layer thread structure
  - Process Layer (Virtual Processors)
    - Linux scheduler is aware of this layer
    - If CPUs are available, multiple virtual processors may run simultaneously
    - Responsible for scheduling and running user layer threads
  - User Layer
    - Individual stack space, context registers, and thread specific data
    - Only Virtual Processors are aware of user layer threads

MLTP Architecture



- The Linux kernel is responsible for assigning virtual processors (VPs) to processors and allowing them to run
- Virtual Processors are responsible for scheduling and running user threads

Process Layer

- This is a layer consists of Virtual Processors created by the clone() function
- Virtual Processors have the following properties
  - Each VP is a process
  - Each VP shares resources like a kernel-level thread
  - Each VP is scheduled by the kernel
- Virtual processors cooperate with each other to schedule and run user threads
  - Currently a round-robin scheduling is implemented

User Layer

- This layer consists of user-level threads, each with the following properties:
  - User-Level threads are not associated with a specific process
    - They may run using their own stack in the context of any Virtual Processor
    - **<u>Exception:</u>** Bound threads are processes existing in the user layer
  - The Linux scheduler is not aware of the existence of unbound threads in the user layer
  - Context switching between threads is cooperative

Implementation of Layering

- Modifications were made to two thread packages to provide a layering
- The Process Layer is based upon jkthreads
  - Uses clone() processes to provide threads
    - Threads are processes sharing, memory space, file system, and file descriptors
    - One extra parent process is required to handle signals from terminating children. At all other times this process sleeps
  - System V semaphores are used for synchronization
  - Tolerant to changes in stack space

Implementation of Layering

- The User Layer is built from QuickThreads
  - Provides a frame work for a fast context switching user-level threads
  - Threads have unique stack space and context registers; they may allocate a thread specific data area on the heap.
  - User-Level threads share, memory space, file system, and file descriptors
    - New spaces are not created, the virtual processors shared resources are used

Thread Scheduling

- Virtual Processors are Linux processes and are scheduled by the Linux kernel
- All ready user-level threads are placed in a single priority run queue
  - Free VPs will attempt to run the first available user-level thread in the run queue
  - When user-level threads yield or abort the VP will begin execution of the next thread in the run queue
- When there are more VPs then user-level threads, free VPs will terminate

User-Level Primitives

- The following classes of primitives are available to user-level threads under MLTP:
  - Thread creation
  - Thread yielding
  - Critical section locking
  - Barrier synchronization
  - Conditional signaling

Thread Creation

- MLTP allows for three different types of threads
  - Unbound single argument, unbound variable argument, and bound single argument
  - There is little difference between unbound single argument and variable argument threads
    - While alive, both thread types behave the same and may make the same function calls
  - Bound threads are created the similarly to Virtual Processors and run on a dedicated process

Unbound Thread Creation

- Two types single and variable arguement
  - `mltp_create(mltp_userf_t *func, void *p0)`
  - `mltp_vcreate(mltp_vuserf_t *func, int nbytes, ...)`
    - Uses vargs to get parameters
- Role
  - Allocates and aligns space for stack and context
  - Places address and parameters of thread's main function on the stack.
  - Puts thread at the end of the run queue

Bound Thread Creation

- Prototype
  - `mltp_create_bound(mltp_buserf_t *func, void *p0, int stacksz, mltp_buserf_t *term)`
- Role
  - Wrapper function calling jkthread's jkthread_create()
  - Uses clone to create a thread processes which immediately begins to run
  - Returns MLTP thread structure for compatibility with MLTP functions

Thread Yielding and Termination

- Since unbound threads cooperatively multi-task, a yielding mechanism is required
- Two types of yielding are provided yielding
  - A thread may place itself at the head or tail of the run queue
  - `mltp_yield(void)`
  - `mltp_yield_to_first(void)`
- After a thread yields, the freed VP will execute the next user-level thread in the run queue

Critical Section Locking

- MLTP currently supports three types of locks
  - Spin locks, yielding locks, and front yielding locks
- Spin locks are ticket style locks
  - Atomic action is required to acquire a ticket
  - Once ticket is acquired spin until ticket is served
- Yielding lock acquisition is atomic
  - Unsuccessful threads yield at head or end of run queue
- Future work may be done using back-off and separate lock queues

Spin Locks

```
volatile unsigned int ticket;

/* get ticket */
for (ticket = lock->next_available;
    !(mltp_compare_and_swap(ticket, (ticket + 1),
        &(lock->next_available)));
    ticket = lock->next_available);

/* spin until ticket is being served */
while (ticket != lock->now_serving);
```

Barrier Synchronization

- MLTP barriers provide a means of waiting until unbound threads reach the same point in the code
  - Currently bound threads may not participate in a barrier
  - MLTP threads reaching a barrier block on the run queue
    - This prevents the need for managing other queues and developing special lock release code
    - Avoids the overhead of conditional waiting
  - Multiple episodes on the same barrier are supported

Barrier Entry (Freeing Thread)

```
/* obtain barrier lock */
mltp_lock(&(barrier->lock));
episode = barrier->episode;

/* increment threads waiting on episode count */
barrier->waiters++;

if (barrier->waiters == count)
{
    /* last thread in barrier start new episode */
    barrier->episode++;
    barrier->waiters = 0;

    mltp_unlock(&(barrier->lock));
}
```

Barrier Entry (Non-Freeing Thread)

```
/* obtain barrier lock */
mltp_lock(&(barrier->lock));
episode = barrier->episode;

/* increment threads waiting on episode count */
barrier->waiters++;

if (barrier->waiters != count)
{
    mltp_unlock(&(barrier->lock));

    while (episode == barrier->episode)
        mltp_yield(); /* yield on run queue */
}
```

Conditional Waiting and Signaling

- Unbound threads may wait (yield) until a conditional event is signaled
  - The current implementation does not support yielding of bound threads
  - Each conditional event has its own yielding queue associated with it
- Any thread may signal or broadcast that an event has occurred
  - Signal puts the first waiting thread on the run queue
  - Broadcast puts all waiting threads on the run queue

Performance Results

- MLTP has been benchmarked using both synthetic and application benchmarks
  - Synthetic benchmarks used to measure context switching cost
  - Applications perform real calculations and require varying amounts of memory and synchronization
    - Pi Approximation - Use little memory and synchronization
    - Matrix Multiplication - Varying memory requirements with little or no synchronization
    - Ocean Simulator - Varying memory requirements with a large amount of synchronization

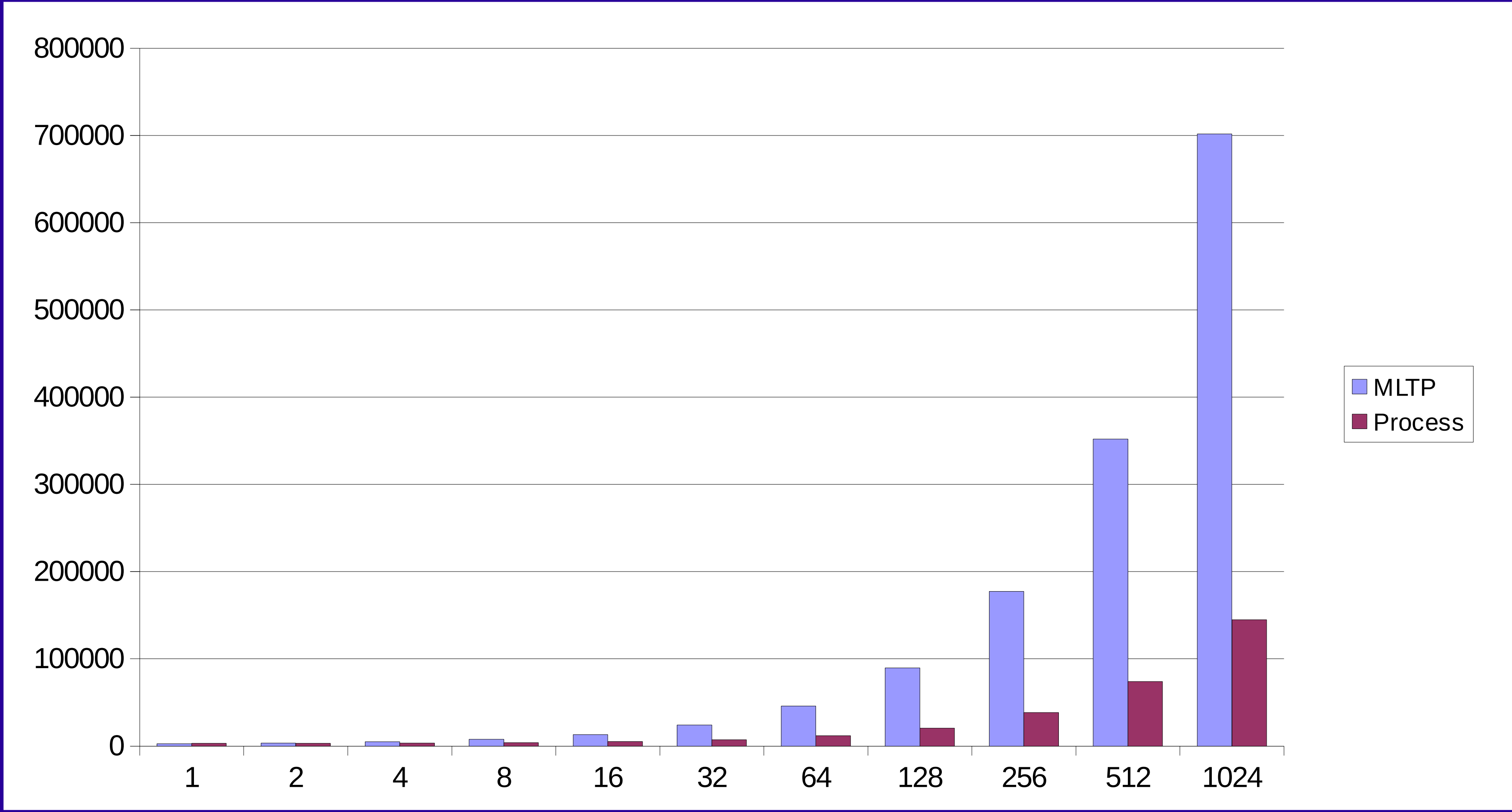
Context Switching Cost

- Project goal was to provide a thread package that operates on multiple processors and has an inexpensive context switch
- Cost of switching between MLTP threads must be measured against the cost of context switching processes
- PCL (Program Counter Library) used to count instruction cycles for both MLTP threads and processes

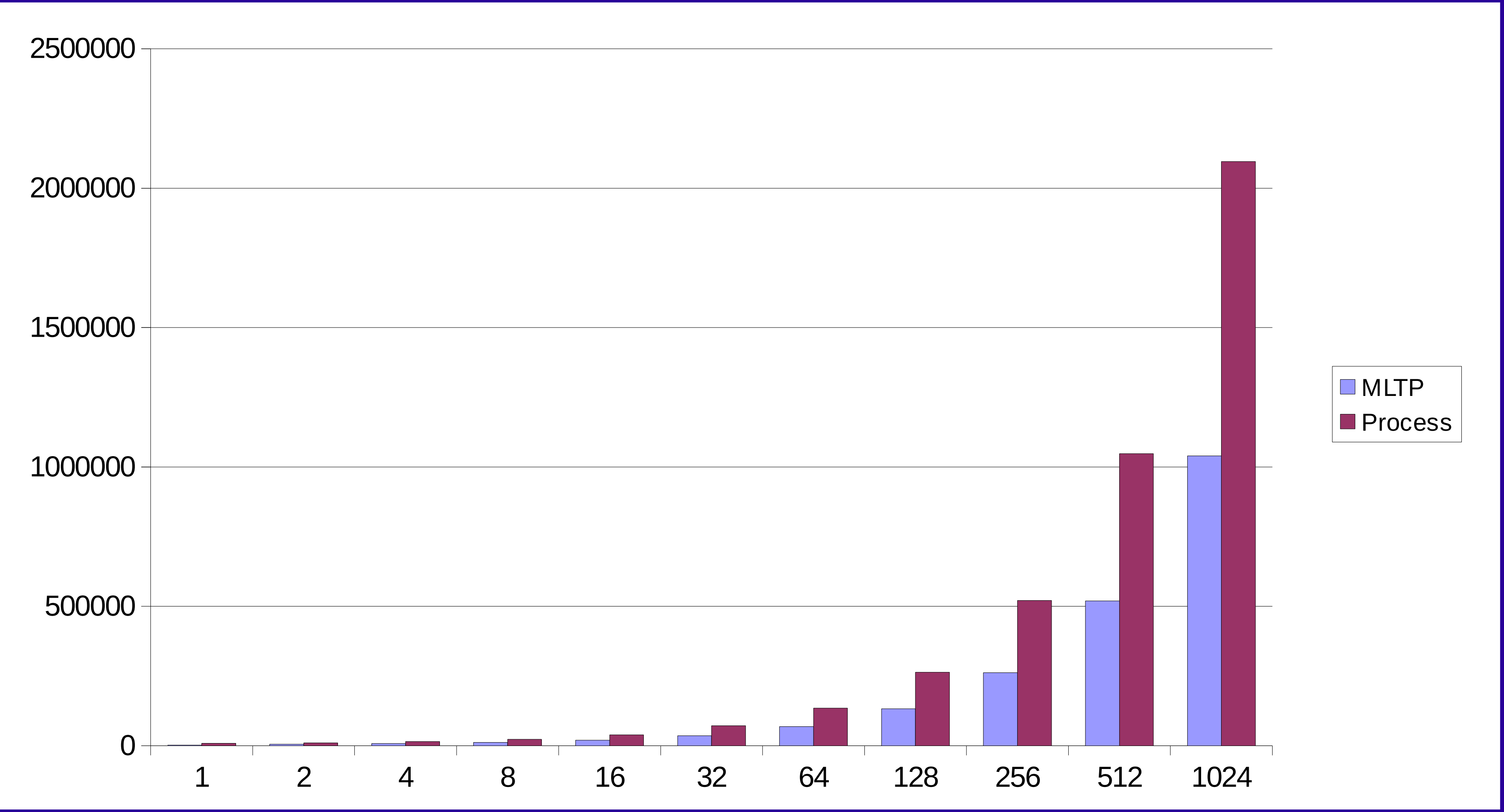
Context Switching Cost

- Measured "light" context switching
  - "Light" context switching is the time it takes to complete a yield instruction
  - Another possibility exists in the case of multi-programmed environments where a thread is started on a processor that is running a program out of a completely different memory space
    - This may cause page buffer and other cache invalidation
- MLTP threads switch context at user level while processes switch context at user and kernel level

Light Context Switch User Cycles



Light Context Switch User and System Cycles



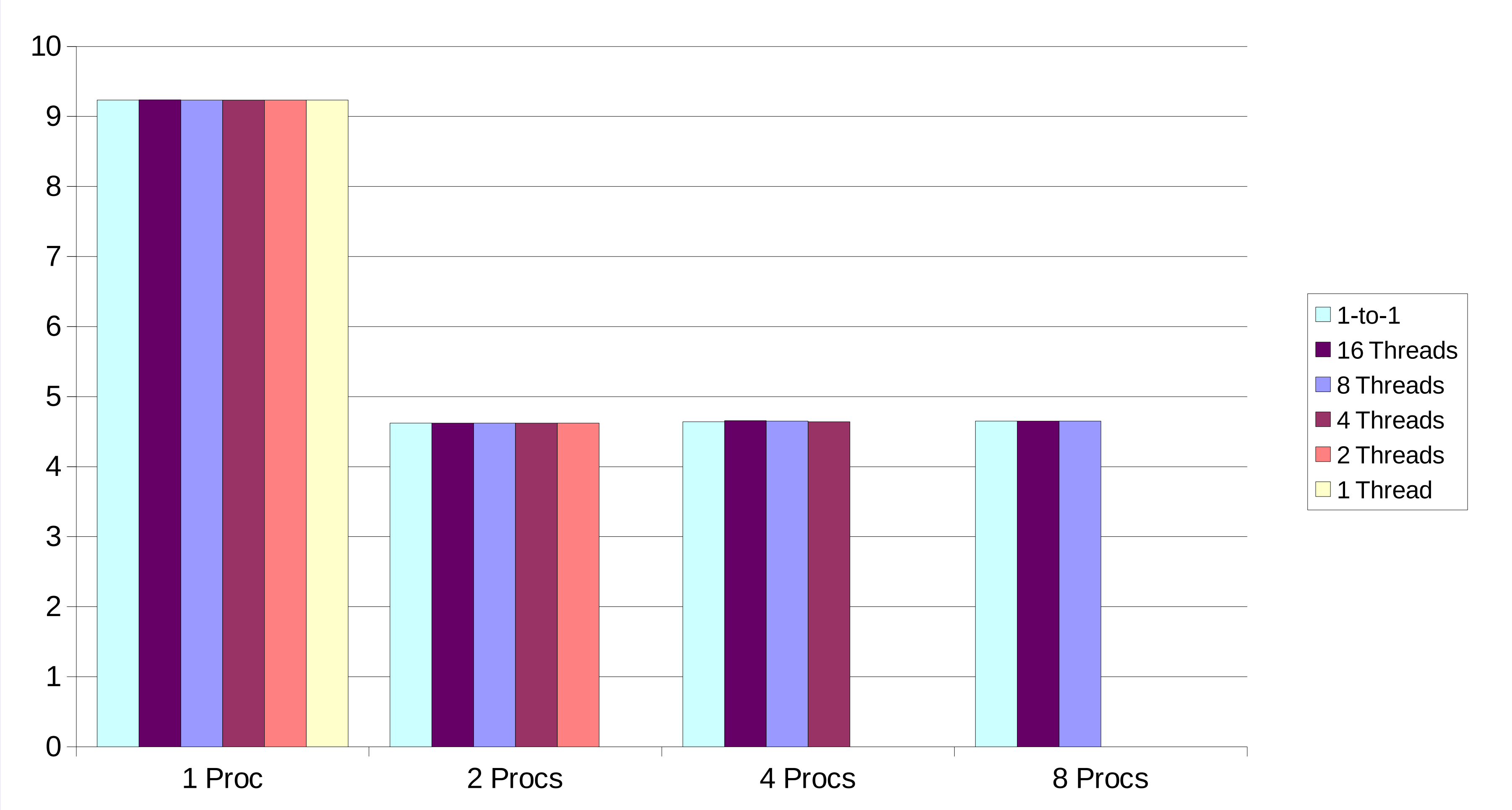
Summary of Context Switch Benchmark

- MLTP thread context switches require a substantially larger amount of user level CPU cycles than processes
  - 282 instructions per switch versus 10
- Total CPU cycles executed by MLTP context switch is approximately half that of processes
  - 378 instructions per switch versus 830
  - 500MHz machine may save about 2ms per 1000 context switches

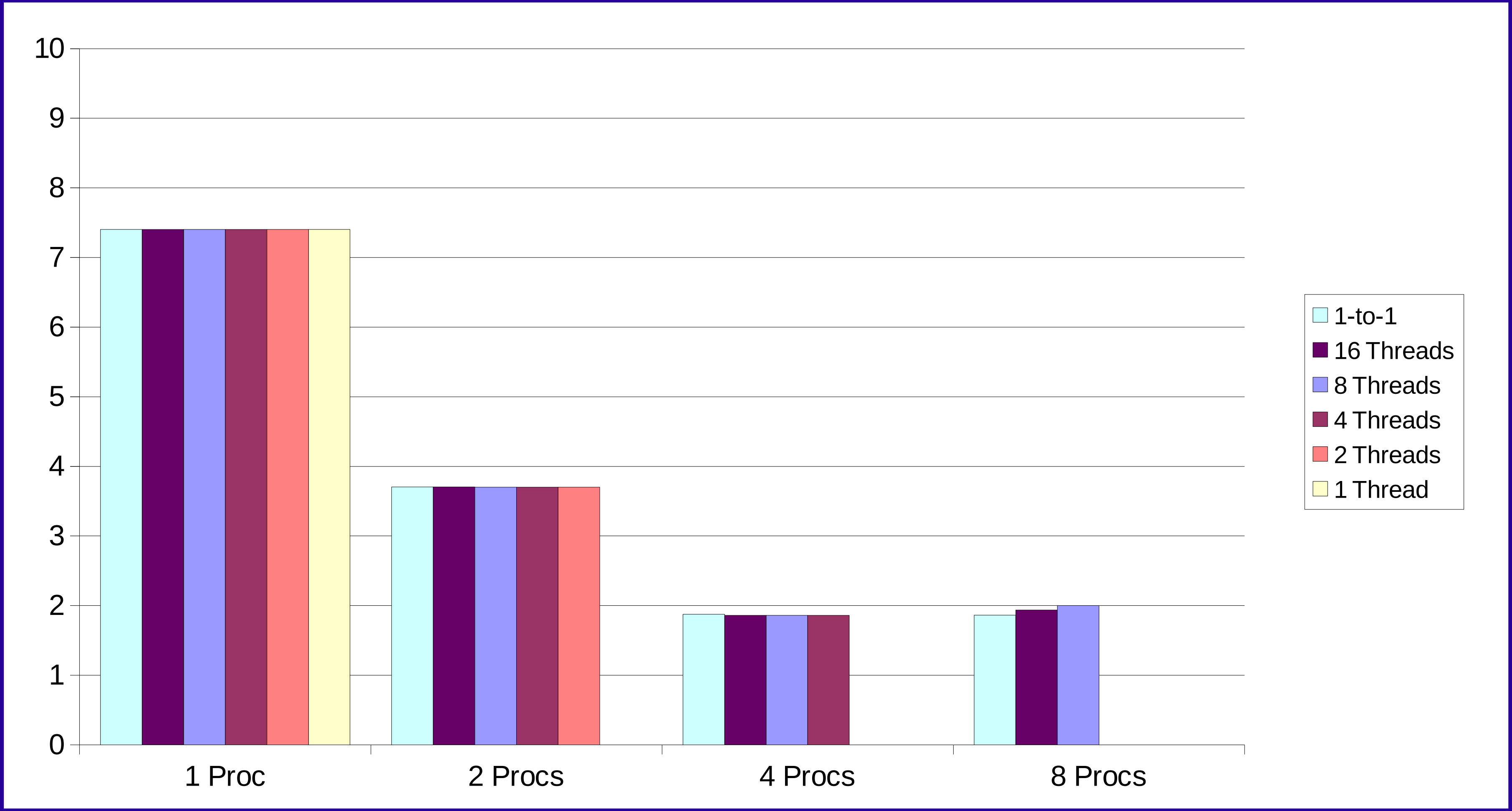
Pi Approximation

- Approximates Pi by using the rectangle rule to compute the area of the unit circle
- Trivially parallelizable
  - Regions summed are divided among threads
  - Locking is only required when each thread's results are added to a sum total
  - No other synchronization required
  - No forced context switches

Pi Approximation 2 Processors



Pi Approximation 4 Processors

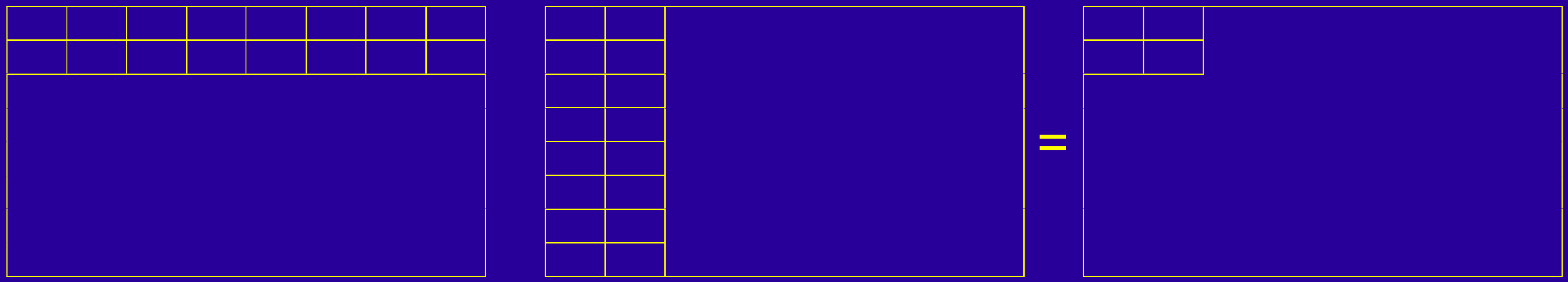


Summary of Pi Benchmark

- User-Level thread count has minimal impact on average execution times
  - The number of processes has a far greater effect
- Best result is usually one MLTP thread per VP per processor
  - Utilizes each processor
  - Minimizes context switching
  - Uses faster, low contention, user-level locks

Matrix Multiplication

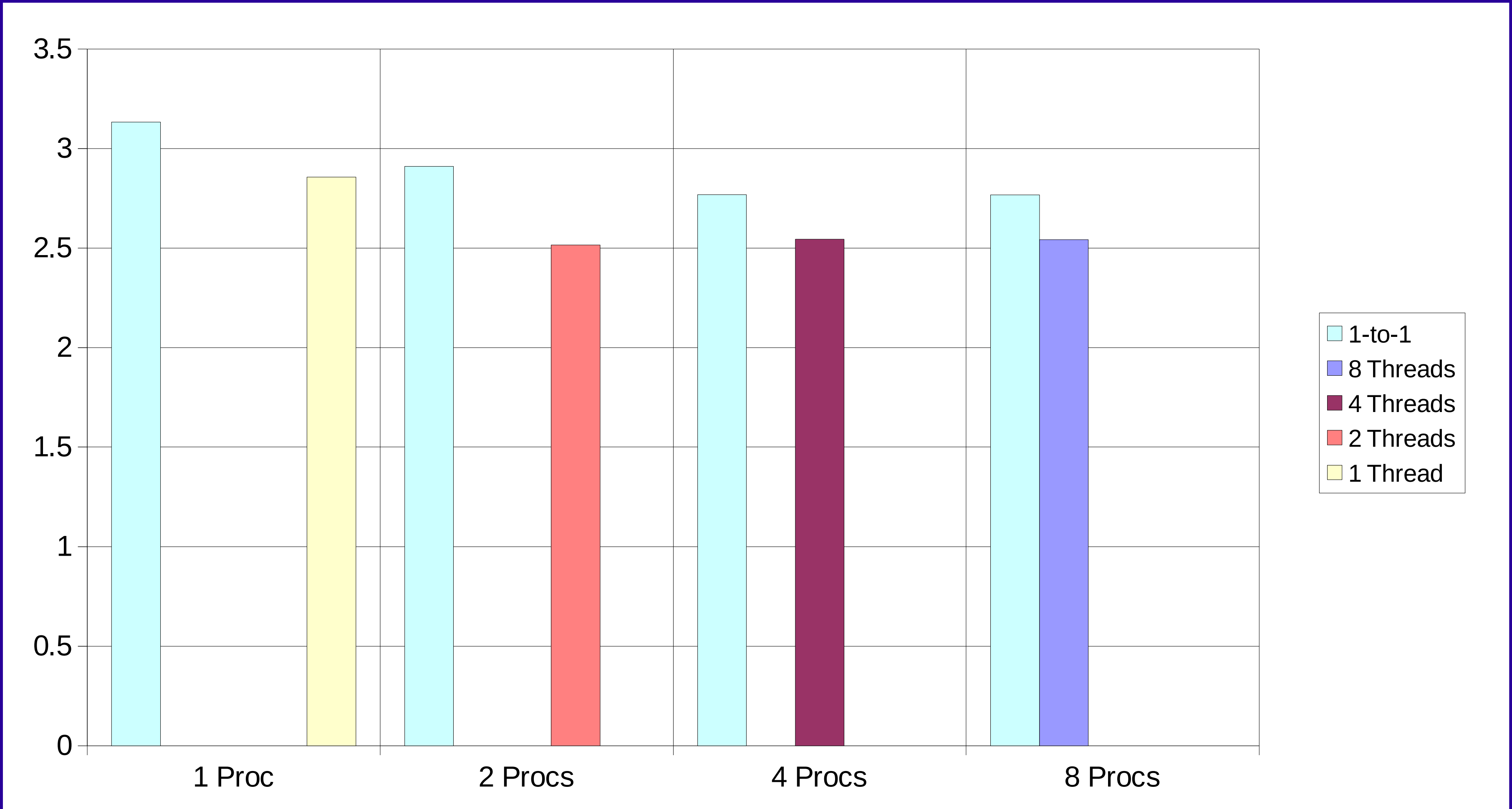
- Use submatrices to Multiply two N×N matrices
- Initially investigated a dynamic self-scheduling of submatrix multiplication
  - Uses one lock for synchronization
    - No yielding, signaling, or barriers



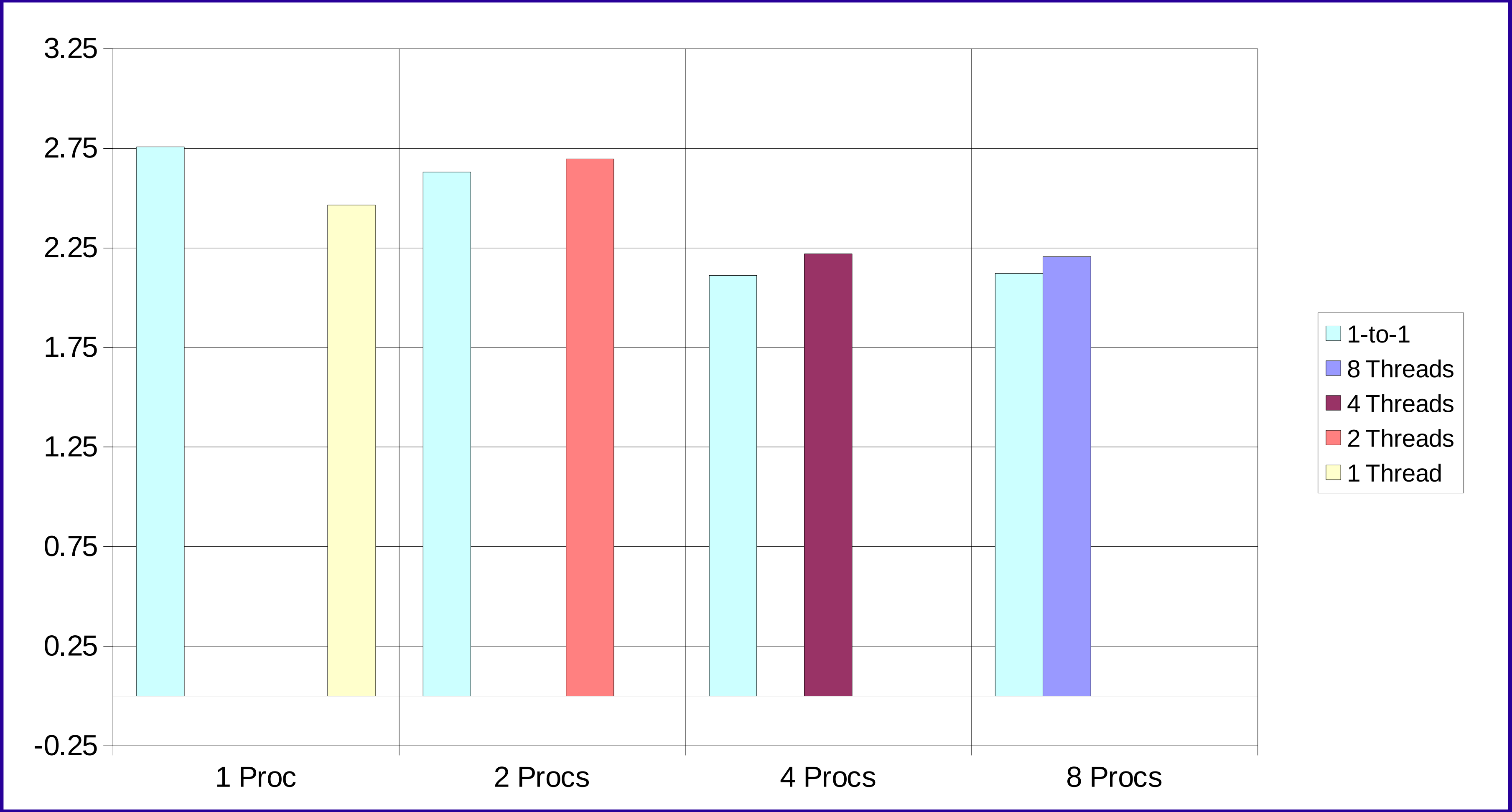
Matrix Multiplication

- For fixed version thread ID determines which submatrices a thread is responsible for
  - No synchronization is required
  - Workload is divided evenly among threads
  - Fairer benchmark with MLTP
    - One user thread does not do all the processing per a process

Dynamic Scheduling 2 Processors



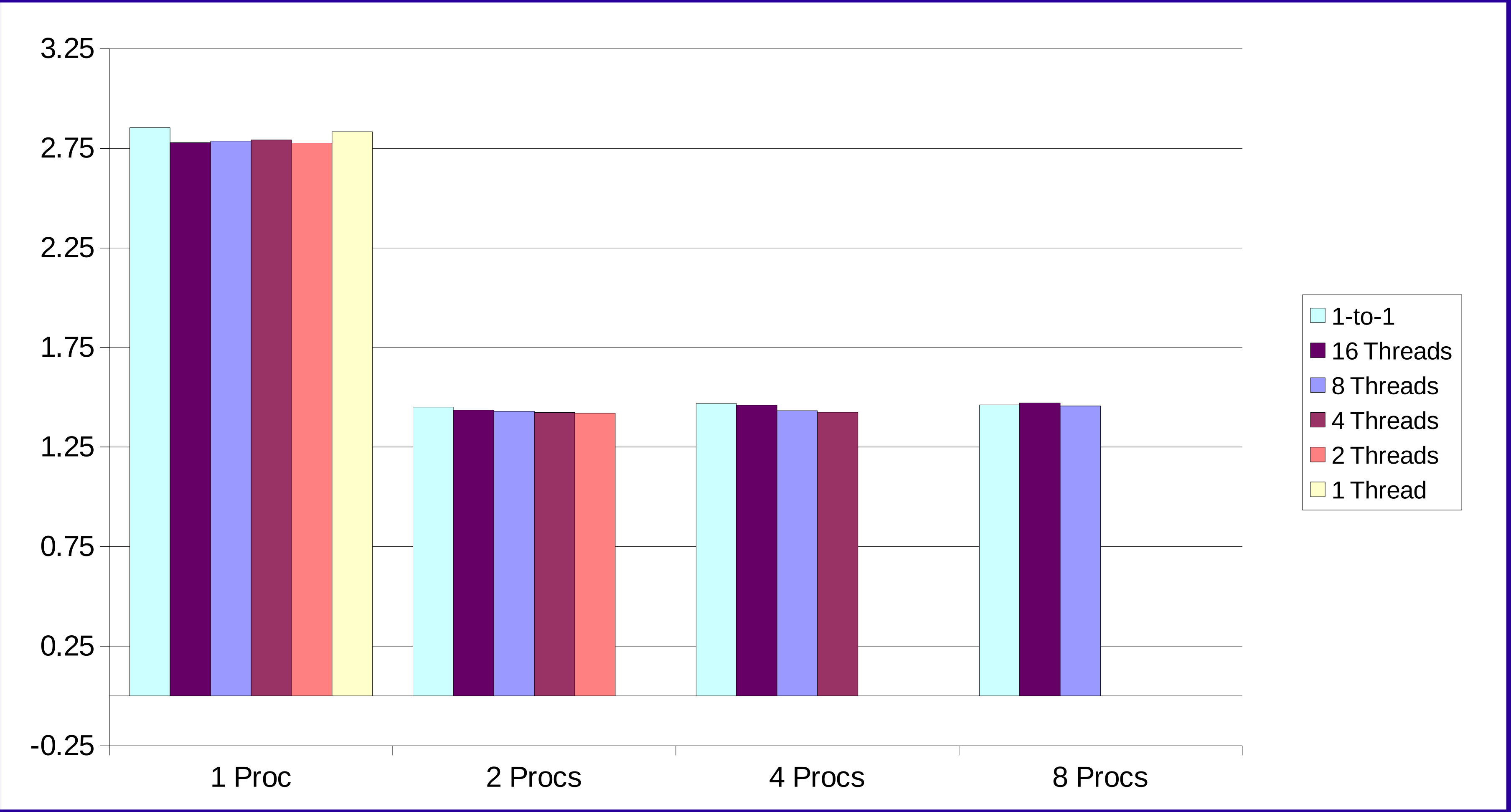
Dynamic Scheduling 4 Processors



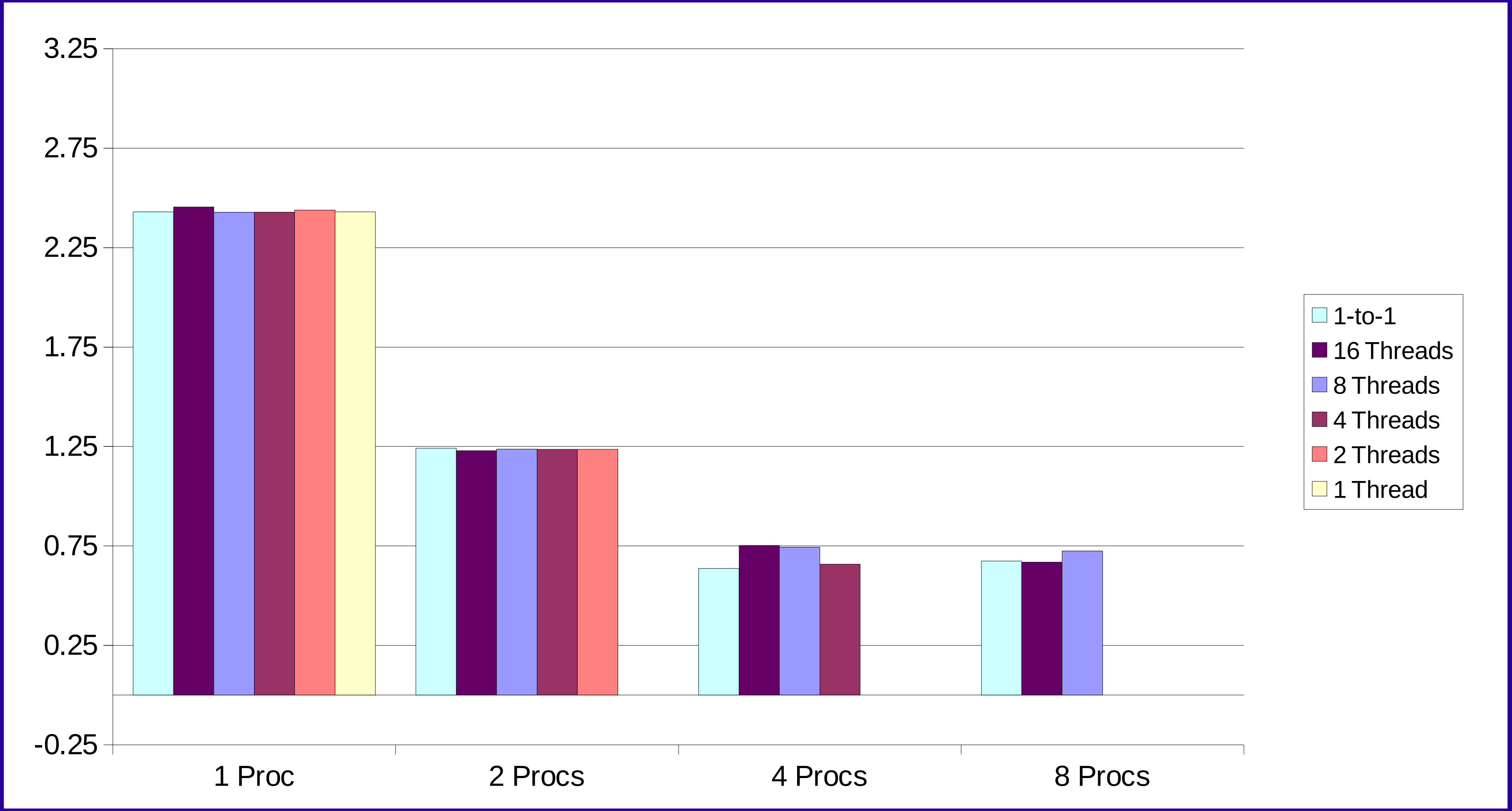
Summary of Dynamically Scheduled Matrix Multiplication Benchmark

- Only meaningful for kernel-level threads or MLTP with one thread per VP
- Small penalty for more processors than processes
- MLTP achieves best results with one thread per VP per processor
- There is no clear performance winner between MLTP and kernel-level threads
  - Minimal synchronization and context switching, should favor kernel-level threads

Fixed Scheduling 2 Processors



Fixed Scheduling 4 Processors



Summary of Fixed Schedule Matrix Multiplication Benchmark

- On two processor machines MLTP runs slightly faster than Kernel-level threads
  - With more processes opposite is true on four processor machines
  - May be related to processor migration of user-level threads
    - Executions with large number of user-level threads and four or more VPs loose efficiency
- Timing difference between thread packages is not substantial

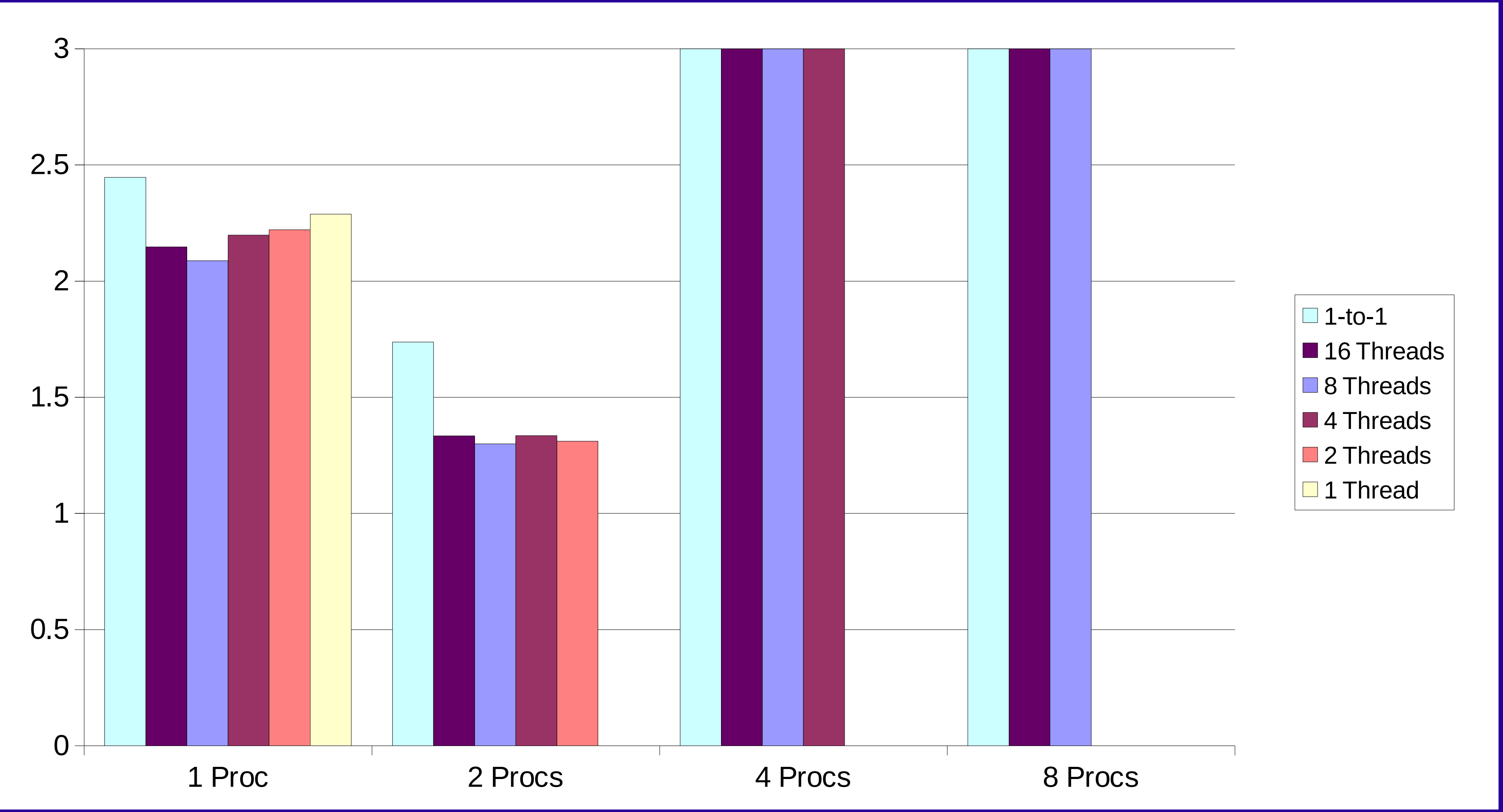
SPLASH-2 Ocean

- Stanford Parallel Applications for Shared Memory Ocean current simulation
- Ocean computes current flow for various layers of an ocean region
- An ocean region is divided into a grid of layers and streamfunction is used to determine how each region effects its neighbor
  - Problem is similar to many other grid solver problems
- Has 10 synchronization phases for computation

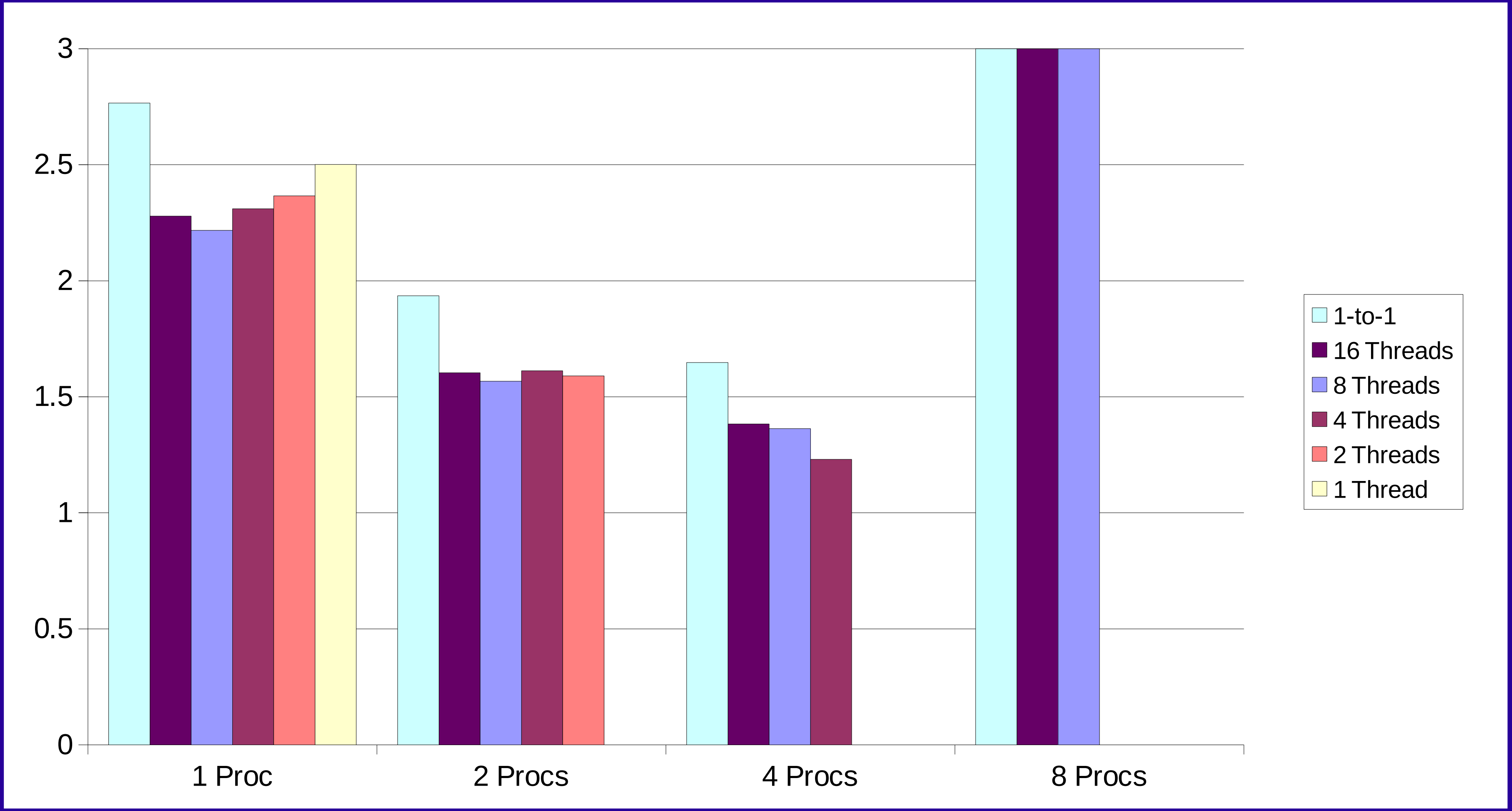
Ocean Synchronization

Put Laplacian of Ψ_L in W1$_L$	Put Laplacian of Ψ_3 in W1$_3$	Copy Ψ_L, Ψ_3 to T_L, T_3	Put $\Psi_L - \Psi_3$ in W2	Put computed Ψ_2 vals in W2	Initialize γ_a and γ_b
Add f values to columns of W1$_L$ and W1$_3$		Copy Ψ_{LM}, Ψ_{3M} into Ψ_L, Ψ_3			Put Lapacian of Ψ_{LM}, Ψ_{3M} in W7$_{L3}$
Put Jacobians of (W1$_L$, T1$_L$), (W1$_3$, T1$_3$) in W5$_l$, W5$_3$		Copy T_L, T_3 into Ψ_{LM}, Ψ_{3M}			Put Lapacian of W7$_{L3}$ in W4$_{L3}$
			Put Jacobian of (W2, W3) in W6		Put Lapacian of W4$_{L3}$ in W7$_{L3}$
Update the γ expressions					
Solve the equations for Ψ_a and put the result in γ_a					
Compute the integral of Ψ_a					
Compute $\Psi = \Psi_a + C(t)\Psi_b$ (Note: Ψ_a and now Ψ are maintained in the γ_a matrix)			Solve the equation for Φ and put result in γ_b		
Use Ψ_a and Φ to update Ψ_L and Ψ_3					
Update streamfunction running sums determine when to stop running program					
Note: Horizontal lines represent synchronization points among all processes, and vertical lines vertical lines spanning phases demarcate threads of dependence.					

258x258 Ocean 2 Processors



258x258 Ocean 4 Processors



Ocean Benchmark Summary

- MLTP substantially out performs kernel-level threads when there is no more than one VP per processor
  - Performance is terrible when there is more than one VP per processor
    - 448 seconds with 4 VPs and two processors
- For many cases 8 user-level threads with a lesser number of VPs had best performance
  - More threads process smaller grids and obtain a greater percent of cache hits

Contribution/Conclusion

- Main contribution: a robust and extensible two-layer thread implementation for SMP Linux, capable of implementing a variety of multi-threading applications
- MLTP provides a less costly context switch and thread synchronization then process based threads
- MLTP does not out perform process based threads on applications which do not require context switching and synchronization