



# A Test of Java™ Virtual Machine Performance

By Ed Ort

(February 2001)

Prior to Version 1.3, production releases of the [Java™ 2 SDK, Standard Edition for the Solaris™ Operating Environment](#) included a virtual machine<sup>1</sup> implementation known as Exact VM (EVM). With the release of [Java 2 SDK, Standard Edition 1.3](#), EVM is replaced by the [Java HotSpot™ Server VM](#) and the [Java HotSpot™ Client VM](#). The Java HotSpot Client and Server VMs are designed to deliver a high level of performance for Java technology-based applications, in many cases a higher level of performance than these applications could achieve with EVM. However, as is true for most performance-enhancing tools, the performance of an application can also depend on other factors such as VM options specified at runtime.

This article presents the results of a test that compared the performance of a Java application running with the JDK 1.2.2\_05a Production Release for Solaris, to the performance of that application running with the Java 2 SDK 1.3 Early Access release for Solaris. The test also measured the impact of various VM option settings on performance. These results can help you determine which version of Java 2 SDK for Solaris is best for your needs, and what JVM settings are best for certain types of applications.

## HotSpot Client and Server VM Distribution

The Java HotSpot Client and Server VMs are available as follows:

- The Solaris and Linux versions of Java 2 SDK, Standard Edition V 1.3 include the Java HotSpot Server VM and the Java HotSpot Client VM.
- The Windows NT version of Java 2 SDK, Standard Edition V 1.3 includes the Java HotSpot Client VM.
- The Java HotSpot Server VM is available as a separate download for Windows NT.

## Hardware Configuration

The performance test was run on the following hardware:

Processor:	Sun Ultra UPA/PCI (4 X UltraSPARC-II 450 Mhz)
System clock frequency:	113 MHz
Memory size:	4096 Mbytes

## Test Program

The performance test program is a multithreaded program named [HeapTest](#). HeapTest uses classes [HeapThread](#), [Barrier](#), and [Node](#).

The HeapTest program iterates through a number of loops of work. In each loop, it performs a set of memory allocation and computation tasks. In the first loop, HeapTest performs memory allocation exclusively. In the next loop, it performs primarily memory allocation and some computation. With each succeeding loop, it lowers the amount of memory allocation activity and raises the amount of computation activity, until the final loop, in which all the work is computation. No matter what mix of tasks it performs, the program equally distributes the work it does across a number of threads. The maximum number of threads HeapTest starts, and the number of loops HeapTest executes, depends on arguments specified at runtime. For example:

```
java heaptest.HeapTest 5 2
```

runs HeapTest (in the heaptest package) with a maximum of 5 threads, and performs a maximum of 2 loops for either memory allocation or computation. The maximum loop value of 2 results in HeapTest executing a set of tasks three times: the first time, it performs two loops of memory allocation; the second time, it performs one loop of memory allocation and one loop of computation; the third time, it performs two loops of computation.

For each run, the program prints the total time to perform the work (that is, memory allocation and computation) for a given number of threads, up to the maximum number of threads available. Here's an example of HeapTest output for a maximum of 5 threads, and a maximum of two loops:

Threads		2 Heap, 0 CPU	1 Heap, 1 CPU	0 Heap, 2 CPU
1	8887	14203	18855	
2	9836	14865	18657	
3	10588	16572	18738	
4	10253	16190	18303	
5	10194	16675	18895	

## Test Configurations

The performance test comprised multiple runs of the HeapTest program with either the Java 2 SDK 1.2.2\_05a production release for the Solaris Operating Environment, or the Java 2 SDK 1.3.0 EA Early Access release for the Solaris Operating Environment. (In this article, these individual runs are referred to as "tests.") In each of these tests, HeapTest executed with two separate combinations of Java 2 SDK and runtime options. Here are the combinations:

- Java 2 SDK 1.2.2\_05a with two different sets of runtime options.
- Java 2 SDK 1.3.0 EA with two different sets of runtime options.
- Java 2 SDK 1.2.2\_05a with one set of runtime options, and Java 2 SDK 1.3.0 EA with another set of runtime options.

If you think of one combination of Java 2 SDK and runtime options for a test as "configuration A", and the other combination of Java 2 SDK and runtime options for a test as "configuration B", then each test captured four types of performance results, as follows:

- The time performing memory-allocation tasks with configuration A.
- The time performing memory-allocation tasks with configuration B.
- The time performing computation tasks with configuration A.
- The time performing computation tasks with configuration B.

The specific combinations of Java 2 SDK production release and runtime options in effect for each test are as follows:

<i>Test Number</i>	<i>Java 2 SDK Production Release</i>	<i>Runtime Options<sup>1 2</sup></i>
<a href="#">Test 1</a>	Java 2 SDK 1.2.2_05a Java 2 SDK 1.2.2_05a	default -Xms256m -Xmx512m -Xgencfg:64m,64m,semispaces:64m,512m,markcompact -Xoptimize
<a href="#">Test 2</a>	Java 2 SDK 1.2.2_05a Java 2 SDK 1.2.2_05a	default -Xms256m -Xmx512m
<a href="#">Test 3</a>	Java 2 SDK 1.2.2_05a Java 2 SDK 1.3.0 EA	default -server
<a href="#">Test 4<sup>3</sup></a>	Java 2 SDK 1.2.2_05a Java 2 SDK 1.3.0 EA	default -server
<a href="#">Test 5</a>	Java 2 SDK 1.3.0 EA Java 2 SDK 1.3.0 EA	-client -server
<a href="#">Test 6</a>	Java 2 SDK 1.3.0 EA Java 2 SDK 1.3.0 EA	-server -server -XX:+UseBoundThreads

<a href="#">Test 7<sup>3</sup></a>	Java 2 SDK 1.2.2_05a Java 2 SDK 1.3.0 EA	-Xms256m -Xmx512m -server -Xms256m -Xmx512m
<a href="#">Test 8</a>	Java 2 SDK 1.2.2_05a Java 2 SDK 1.3.0 EA	-Xms256m -Xmx512m -Xgenconfig:64m,64m,semispaces:64m,512m,markcompact -Xoptimize -server -XX:newSize=128m -XX:MaxNewSize=128m -Xms256m -Xmx512m -XX:SurvivorRatio=2
<a href="#">Test 9</a>	Java 2 SDK 1.3.0 EA Java 2 SDK 1.3.0 EA	-server -server -Xnoclassgc
<a href="#">Test 10</a>	Java 2 SDK 1.3.0 EA Java 2 SDK 1.3.0 EA	-server -server -Xnconcurrentio

<sup>1</sup>Runtime options that begin with -X are non-standard. They are not guaranteed to be supported on all VM implementations, and are subject to change without notice in subsequent releases of the Java 2 SDK.

<sup>2</sup>Options that begin with -XX are generally intended for Sun's internal development use. Sun is not committed to supporting these options by fixing any bugs noted with them; these options are subject to change without notice. You use these options at your own risk.

<sup>3</sup>In this test the environment variable LD\_LIBRARY\_PATH was set to /usr/lib/lwp

## Test 1

In this test, HeapTest ran with:

- Java 2 SDK 1.2.2\_05a with no runtime options specified.
- Java 2 SDK 1.2.2\_05a with the runtime options -Xms256m -Xmx512m -Xgenconfig:64m,64m,semispaces:64m,512m,markcompact -Xoptimize

### *Option*

-Xms256m

-Xmx512m

### *Meaning*

Set the initial size of the Java memory allocation pool (that is, the heap) to 256 Mbytes

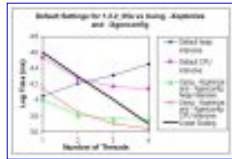
Set the maximum heap size to 512 Mbytes

-Xgencfg:64m,64m,semispaces:64m,512m,markcompact

Configure the heap for generational garbage collection, as follows:  
 Allocate a young generation of two 64 Mbytes semispaces  
 (64m,64m,semispaces)  
 Allocate an initial object nursery space of 64 Mbytes, with a maximum memory allocation of 512 Mbytes;  
 mark-compact the old generation  
 (64m,512m,markcompact)  
 Use the optimizing JIT compiler

-optimize

The results of the test are illustrated in the following graph:



*Click to enlarge*

The results indicate that the performance of an application that does primarily memory-allocation tasks (marked in the graph as "heap-intensive") or computation tasks (marked in the graph as "CPU-intensive") improves if the heap is configured and the optimizing JIT compiler is specified. The optimizing JIT compiler especially improves the performance of computation-intensive code.

Note that a number of runtime specifications are pertinent to generational garbage collection. In generational garbage collection, all new objects are allocated from a "nursery" (also known as the eden space or young space). All the objects in the nursery constitute a "young generation" of objects. When the nursery is full, the garbage collector does a partial garbage collection. It reclaims memory in the nursery for objects that are no longer accessible, that is, "dead" objects. Objects in the nursery that are still "live" get moved to an area of memory for older objects (the "old generation"). Generational garbage collection can be much faster than full garbage collection because the garbage collector does not have to search all of memory for dead objects.

The runtime specification -Xms256m -Xmx512m

-Xgencfg:64m,64m,semispaces:64m,512m,markcompact -Xoptimize results in an initial heap size of 256 Mbytes. Of that 256 Mbytes, 128 Mbytes are allocated for the nursery. The nursery is subdivided into two 64 Mbyte "semispaces." When the youngest semispace is full, the garbage collector reclaims memory in the semispace, and moves any live objects in the semispace to the other semispace. When the older semispace is full, the garbage collector reclaims memory there, and moves any live objects to old-object memory. The markcompact options specifies that when old-object memory needs to be reclaimed, the garbage collector follows a mark-compact algorithm. In this approach the garbage collector compacts gaps left by dead objects in the "tree" of live objects. Compacting gaps in the tree also tends to make object allocation faster.

Some things to notice:

- The graph displays a line labeled "linear scaling." This is a trend line that plots the ideal slope in multithreaded performance. In general, multithreaded code cannot improve on the downward slope of the linear scaling line. Graphs for the other tests in this article also show the linear scaling line.
- Performance got worse when the number of threads increased for the default heap-intensive run. That's because as the number of threads

increased, the contention for the heap lock increased. The heap lock is a monitor that controls synchronized access to the heap.

- Performance improved when the number of threads increased for the default CPU-intensive run. This indicates that multi-threaded code that is CPU-intensive executes better on a multi-processor system.
- There was a cross over in performance for the CPU-intensive and heap-intensive runs using `-Xgencfg` and `-Xoptimize`. That's because with fewer threads, there was less contention for the heap lock. As the number of threads increased, the contention increased. This lead to more time spent in allocation and garbage collection. As the number of threads increased for the CPU-intensive runs, the program was able to utilize the processors more efficiently.

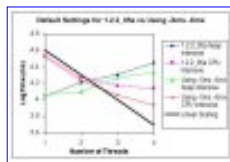
## Test 2

In this test, HeapTest ran with:

- Java 2 SDK 1.2.2\_05a with no runtime options specified.
- Java 2 SDK 1.2.2\_05a with the runtime options `-Xms256m -Xmx512m`

Option	Meaning
<code>-Xms256m</code>	Set the initial size of the Java heap to 256 Mbytes
<code>-Xmx512m</code>	Set the maximum heap size to 512 Mbytes

The results of the test are illustrated in the following graph:



[Click to enlarge](#)

The results indicate that the performance of an application that does primarily heap-intensive or computation tasks improves if the heap is configured, that is, by specifying at run time an initial and maximum heap size. However the performance does not improve as much as it did in [Test 1](#) where the `-Xgencfg` and `-Xoptimize` runtime options were specified.

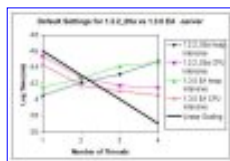
## Test 3

In this test, HeapTest ran with:

- Java 2 SDK 1.2.2\_05a with no runtime options specified.
- Java 2 SDK 1.3.0 EA with the runtime option `-server`

Option	Meaning
<code>-server</code>	Run the JVM in server mode

The results of the test are illustrated in the following graph:



[Click to enlarge](#)

When run with `-server` runtime option, the Java 2 SDK 1.3.0 EA invokes the Java HotSpot Server VM (as opposed to the Java HotSpot Client VM). The two virtual machines are essentially two different just-in-time compilers (JITs) that interface with the same runtime system.

The results of this test appear to indicate that Java 2 SDK 1.2.2\_05a with default settings is more efficient than Java 2 SDK 1.3.0 EA with the `-server` runtime option for applications doing primarily heap-intensive tasks. However, Java 2 SDK 1\_3\_0b27 with the `-server` runtime option is more efficient for applications doing primarily CPU-intensive tasks. Note that the `-server` option in Java 2

SDK 1.3.0 is equivalent to the `-Xoptimize` option in Java 2 SDK 1.2.2. The `-server` option is recommended for all server-side applications, and is especially useful in optimizing CPU-intensive code.

## Test 4

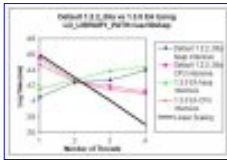
In this test, HeapTest ran with:

- Java 2 SDK 1.2.2\_05a with no runtime options specified.
- Java 2 SDK 1.3.0 EA with the runtime option `-server`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode

In addition, the environment variable `LD_LIBRARY_PATH` was set to `/usr/lib/lwp`

The results of the test are illustrated in the following graph:



*Click to enlarge*

The `LD_LIBRARY_PATH` environment variable specifies the shared libraries path. Setting it to `/usr/lib/lwp` forces the JVM to use a one-level thread model. In a one-level thread model, each thread is associated on a one-to-one basis with the Solaris kernel thread, or Solaris lightweight process (LWP). By default, the JVM uses a two-level model, where threads are multiplexed over possibly fewer LWPs. Using a one-level thread model can produce better throughput for applications because it results in less contention on synchronization objects.

Comparing the results of this test to the results of [Test 3](#), it does not appear that setting `LD_LIBRARY_PATH` to `/usr/lib/lwp` had a significant impact on performance.

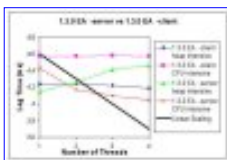
## Test 5

In this test, HeapTest ran with:

- Java 2 SDK 1.3.0 EA with the runtime option `-client`
- Java 2 SDK 1.3.0 EA with the runtime option `-server`

<i>Option</i>	<i>Meaning</i>
<code>-client</code>	Run the JVM in client mode
<code>-server</code>	Run the JVM in server mode

The results of the test are illustrated in the following graph:



*Click to enlarge*

When run with `-client` runtime option, the Java 2 SDK 1.3.0 EA invokes the Java HotSpot Client VM. This VM is optimal for applications that need to start up quickly or that require a small memory footprint. When run with `-server` runtime option, the Java 2 SDK 1.3.0 EA invokes the Java HotSpot Server VM. This VM is optimal for applications that require high performance.

## Test 6

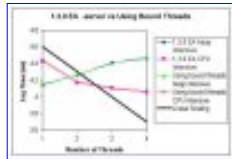
In this test, HeapTest ran with:

- Java 2 SDK 1.3.0 EA with the runtime option `-server`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
- Java 2 SDK 1.3.0 EA with the runtime options `-server -XX:+UseBoundThreads`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
<code>-XX:+UseBoundThreads</code>	Bind user threads to Solaris kernel threads

The results of the test are illustrated in the following graph:



*Click to enlarge*

The runtime option `-XX:+UseBoundThreads` binds each thread to a Solaris kernel thread using the default thread library. In cases where an application uses few threads, the `-XX:+UseBoundThreads` runtime option can help alleviate thread "starvation," a condition where threads don't progress because they don't get enough processor time.

The results of this test do not indicate any difference in the performance of HeapTest when run with or without the `-XX:+UseBoundThreads` runtime option.

## Test 7

In this test, HeapTest ran with:

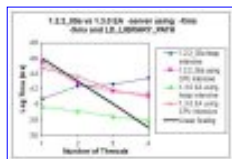
- Java 2 SDK 1.2.2\_05a with the runtime options `-Xms256m -Xmx512m`

<i>Option</i>	<i>Meaning</i>
<code>-Xms256m</code>	Set the initial size of the Java heap to 256 Mbytes
<code>-Xmx512m</code>	Set the maximum heap size to 512 Mbytes
- Java 2 SDK 1.3.0 EA with the runtime options `-server -Xms256m -Xmx512m`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
<code>-Xms256m</code>	Set the initial size of the Java heap to 256 Mbytes
<code>-Xmx512m</code>	Set the maximum heap size to 512 Mbytes

In addition, the environment variable `LD_LIBRARY_PATH` was set to `/usr/lib/lwp`

The results of the test are illustrated in the following graph:



*Click to enlarge*

Comparing the results of this test to the results of [Test 4](#), it appears that with Java 2 SDK 1.3.0 EA, setting the initial and maximum heap size values along with `LD_LIBRARY_PATH=/usr/lib/lwp`, can improve the performance of an application performing heap-intensive tasks. For applications with fewer threads, the alternate thread library gives better throughput. As mentioned in [Test 4](#), `/usr/lib/lwp` forces the JVM to use a one-level thread model. This model results in less contention on synchronization objects.

## Test 8

In this test, HeapTest ran with:

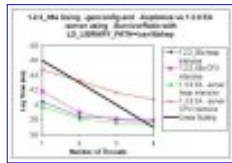
- Java 2 SDK 1.2.2\_05a with the runtime options `-Xms256m -Xmx512m -Xgenconfig:64m,64m,semispaces:64m,512m,markcompact -optimize`

<i>Option</i>	<i>Meaning</i>
---------------	----------------

-Xms256m	Set the initial size of the Java heap to 256 Mbytes
-Xmx512m	Set the maximum heap size to 512 Mbytes
-Xgenconfig:64m,64m,semispaces:64m,512m,markcompact	Configure the heap for generational garbage collection, as follows: Allocate a young generation of two 64 Mbytes semispaces (64m,64m,semispaces) Allocate an initial object nursery space of 64 Mbytes, with a maximum memory allocation of 512 Mbytes; mark-compact the old generation (64m,512m,markcompact)
-optimize	Use the optimizing JIT compiler
<ul style="list-style-type: none"> <li>● <b>Java 2 SDK 1.3.0 EA with the runtime options</b> -server -XX:newSize=128m -XX:MaxNewSize=128m -Xms256m -Xmx512m -XX:SurvivorRatio=2</li> </ul>	
<b>Option</b>	<b>Meaning</b>
-server	Run the JVM in server mode
-XX:newSize=128m	Set the default size for new generation to 128 Mbytes
-XX:MaxNewSize=128m	Set the maximum size for new generation to 128 Mbytes
-Xms256m	Set the initial size of the Java heap to 256 Mbytes
-Xmx512m	Set the maximum heap size to 512 Mbytes
-XX:SurvivorRatio=2	Set the ratio of nursery-to-survivor space to 2

In addition, the environment variable LD\_LIBRARY\_PATH was set to /usr/lib/lwp

The results of the test are illustrated in the following graph:



*Click to enlarge*

In this test, HeapTest ran with the same Java 2 SDK 1.2.2\_05a configuration settings as used in [Test 1](#). It then ran against an equivalent Java 2 SDK 1.3.0 EA configuration. In addition, the environment variable LD\_LIBRARY\_PATH was set to /usr/lib/lwp.

The -XX:NewSize=128m and -XX:MaxNewSize=128m specifications set the initial nursery size to 128 Mbytes, and the maximum nursery size to 128 Mbytes, respectively. The -XX:SurvivorRatio=2 specification sets the ratio of nursery space to "survivor" space to 2. Java 2 SDK 1\_3\_0b27 uses three spaces for young objects, an eden space and two semispaces. All new objects are initially allocated in the eden space. During garbage collection, surviving objects are moved to a semispace. The "survivor space" is the combined size of the two semispaces. This means that for 128 Mbyte nursery, and a survivor ratio of 2, the survivor space is 64 Mbytes and the other two semispaces are 32 Mbytes each. The eden space is 64 Mbytes, that is, the difference between the nursery size and the survivor space.

The results of the test indicate a slight improvement in performance of HeapTest when performing heap-intensive tasks with Java 2 SDK 1\_3\_0b27 and the accompanying VM options and LD\_LIBRARY\_PATH settings, as compared to performing those tasks with Java 2 SDK 1.2.2\_05a and comparable settings.

## Test 9

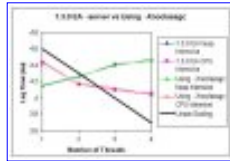
In this test, HeapTest ran with:

- Java 2 SDK 1.3.0 EA with the runtime option `-server`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
- Java 2 SDK 1.3.0 EA with the runtime option `-server -Xnoclassgc`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
<code>-Xnoclassgc</code>	Prevent class unloading during full garbage collection

The results of the test are illustrated in the following graph:



*Click to enlarge*

This test examined the impact of disabling class garbage collection on the performance of an application running with Java 2 SDK 1.3.0 EA and the Java HotSpot Server VM. The runtime option `-Xnoclassgc` disables unloading of unreferenced classes during a full garbage collection. The results indicate that specifying `-Xnoclassgc` has no impact on performance.

## Test 10

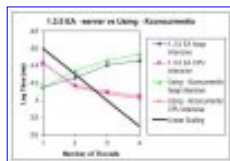
In this test, HeapTest ran with:

- Java 2 SDK 1.3.0 EA with the runtime option `-server`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
- Java 2 SDK 1.3.0 EA with the runtime option `-server -Xconcurrentio`

<i>Option</i>	<i>Meaning</i>
<code>-server</code>	Run the JVM in server mode
<code>-Xconcurrentio</code>	Set the synchronization implementation for concurrent Input/Output

The results of the test are illustrated in the following graph:



*Click to enlarge*

This test examined the impact of the Java 2 SDK 1.3.0 EA runtime option `-Xconcurrentio` on performance. Specifying `-Xconcurrentio` changes the implementation of thread synchronization in a way that sometimes improves throughput for programs having many threads blocking Input/Output. The gains are due to a higher level of concurrency in the Solaris operating system kernel.

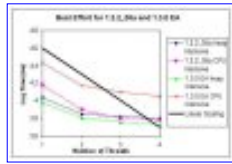
The results indicate that for HeapTest, specifying `-Xconcurrentio` had a slightly positive impact on the performance of heap-intensive tasks. Specifying `-Xconcurrentio` did not improve the performance of CPU-intensive tasks.

## Conclusion

The runtime option settings that produced the best performance results for HeapTest are:

<i>Java 2 SDK Production Release</i>	<i>Runtime Options</i>
Java 2 SDK 1.2.2_05a	-Xms256m -Xmx512m -Xgconfig:64m,64m,semispaces:64m,512m,markcompact -Xoptimize
Java 2 SDK 1.3.0 EA (heap-intensive)	-server -XX:newSize=128m -XX:MaxNewSize=128m -Xms256m -Xmx512m -XX:SurvivorRatio=2
Java 2 SDK 1.3.0 EA (CPU-intensive)	-server

These best performance results are illustrated in the following graph:



*Click to enlarge*

Specifying `-Xgconfig` and `-Xoptimize` yields in the best performance for both heap-intensive and CPU-intensive tasks with Java 2 SDK 1.2.2\_05a. Specifying `SurvivorRatio` yields the best performance for heap-intensive tasks with Java 2 SDK 1.3.0 EA. Specifying `-server` yields the best performance for CPU-intensive tasks with Java 2 SDK 1.3.0 EA.

## For More Information

[Java™ HotSpot VM Options.](#)

[Frequently Asked Questions About The Java™ HotSpot Virtual Machine.](#)

[The Java HotSpot™ Performance Engine: An In-Depth Look.](#)

**Ed Ort** is a staff member of the Java Developer Connection<sup>SM</sup>. He has written extensively about relational database technology and programming languages.

## Reader Feedback

Tell us what you think of this article, and if you would like to see more like it.



Very worth reading      Worth reading      Not worth reading

If you have other comments or ideas for future articles, please type them here:

Have a question about programming? Use [Java Online Support](#).

<sup>1</sup> As used on this web site, the terms Java virtual machine or Java VM mean a virtual machine for the Java platform.

Printable Page 

[ This page was updated: 13-Feb-2001 ]

---

[Products & APIs](#) | [Developer Connection](#) | [Docs & Training](#) | [Online Support](#)  
[Community Discussion](#) | [Industry News](#) | [Solutions Marketplace](#) | [Case Studies](#)

---

[Glossary](#) | [Feedback](#) | [A-Z Index](#)

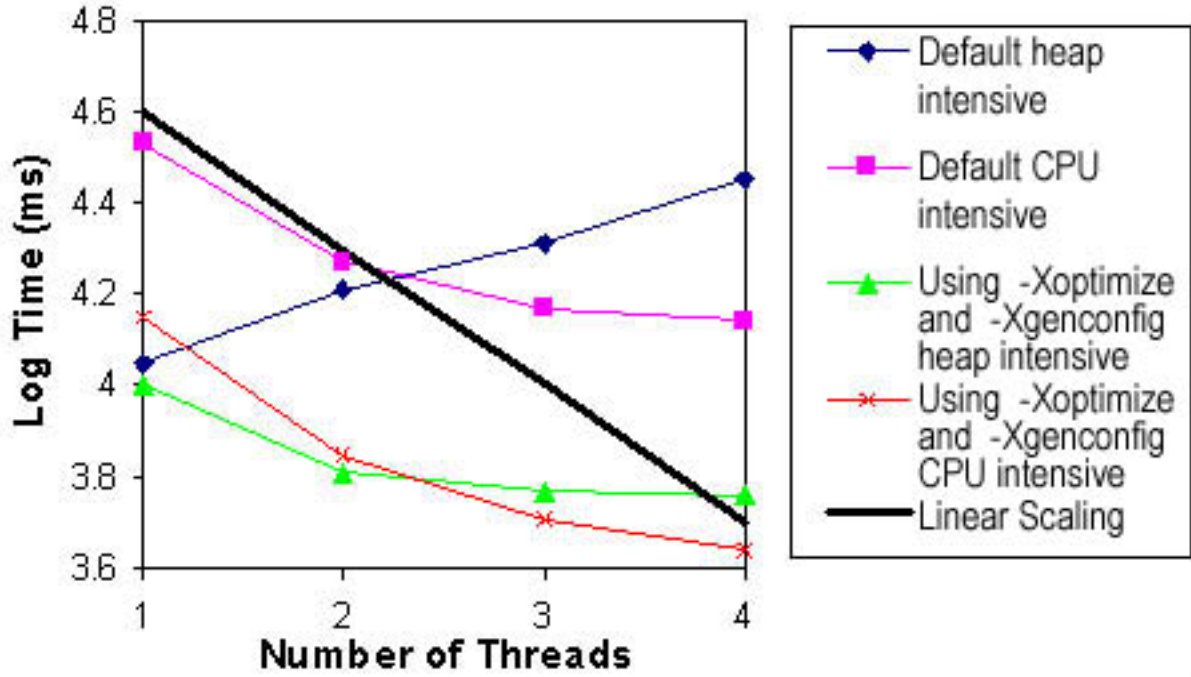
For more information on Java technology  
and other software from Sun Microsystems, call:  
(800) 786-7638

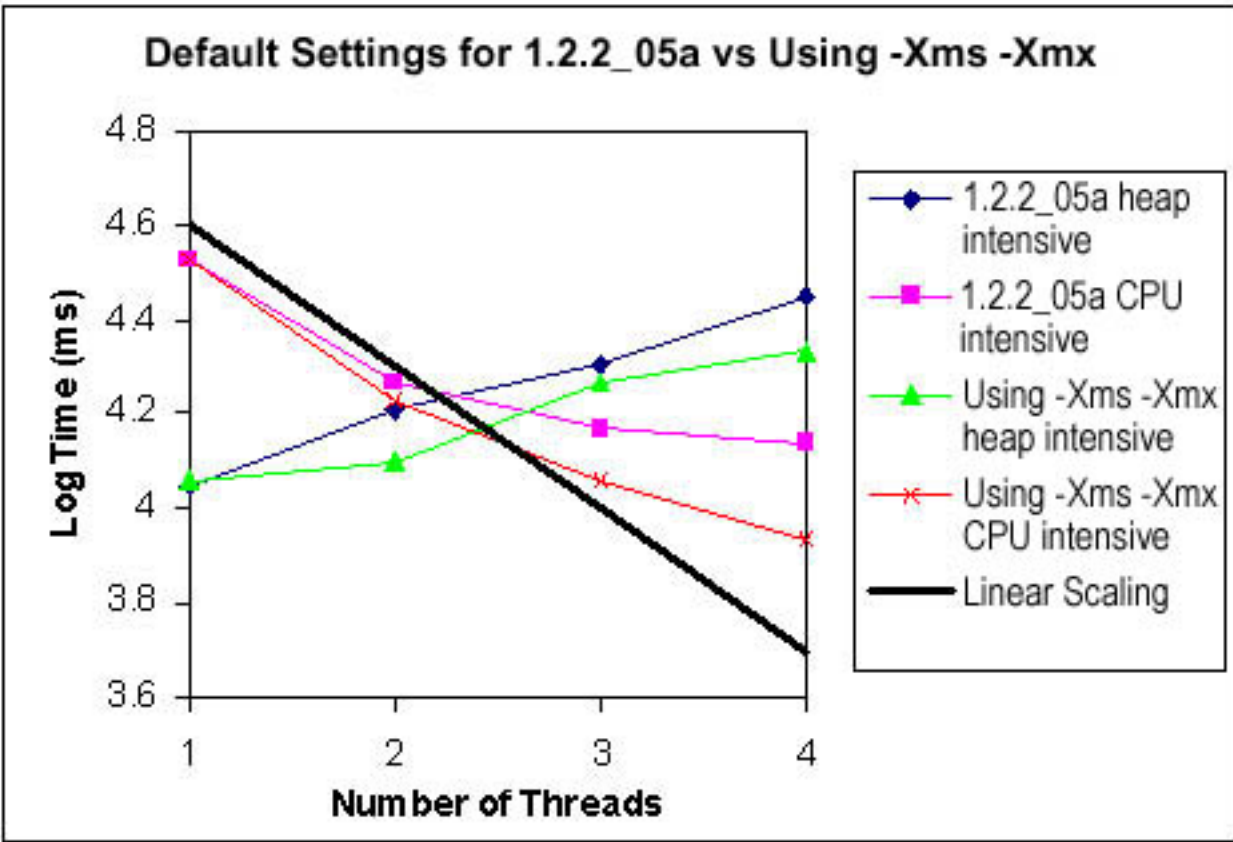
Outside the U.S. and Canada, dial your country's [AT&T Direct Access Number](#)  
first.

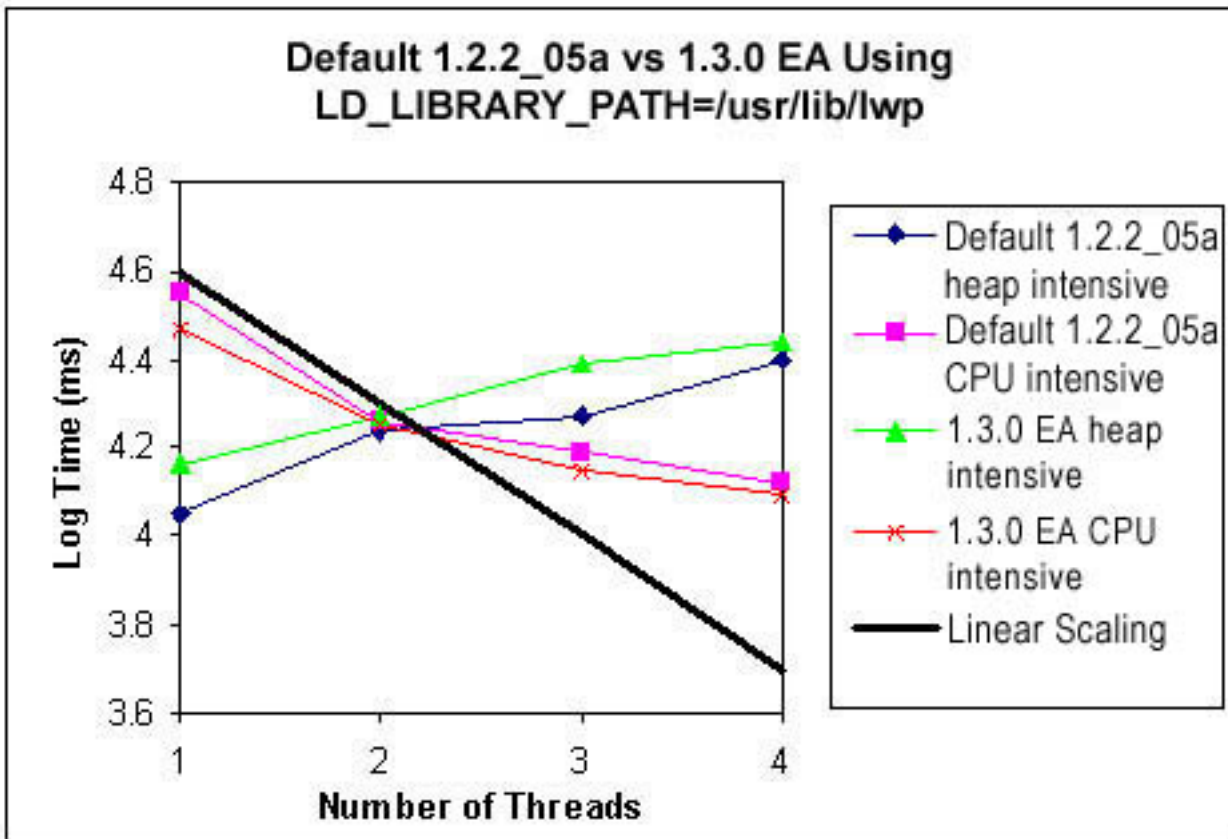


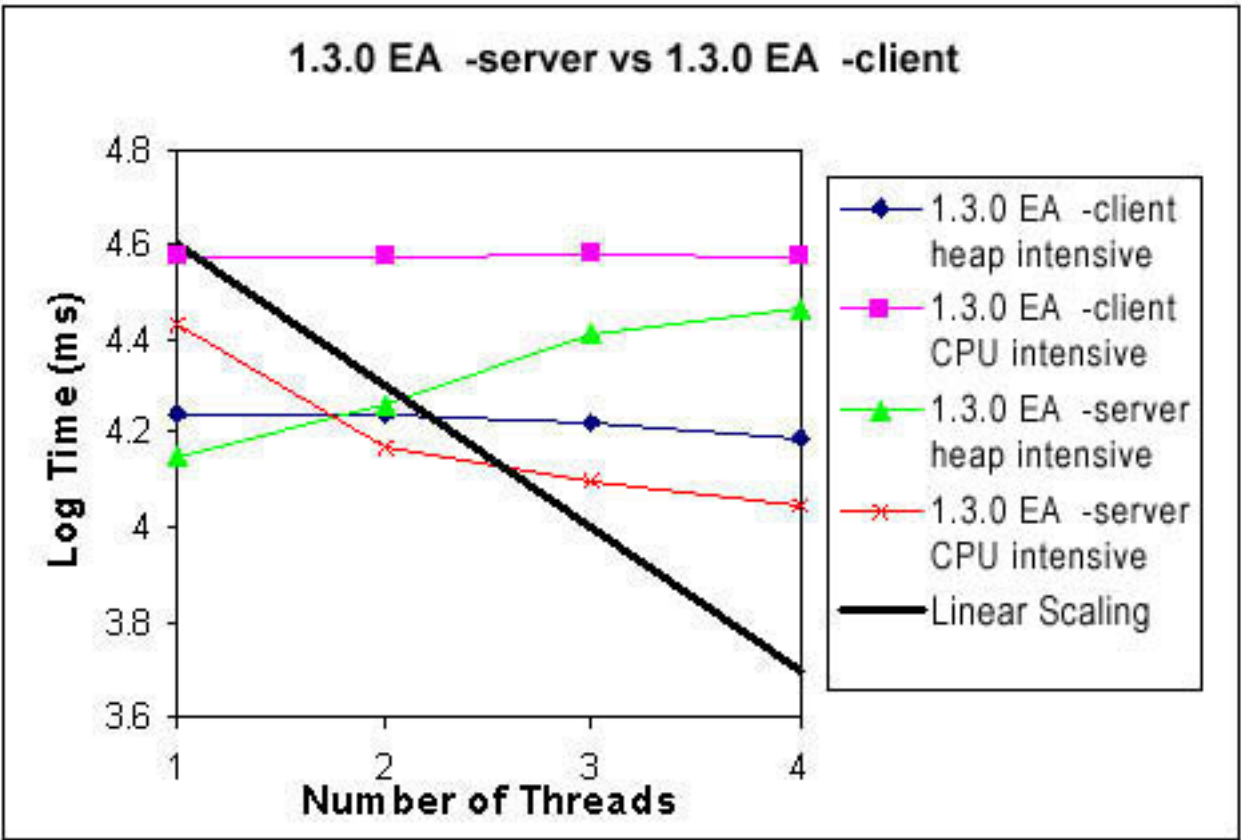
Copyright © 1995-2001 [Sun Microsystems, Inc.](#)  
All Rights Reserved. [Terms of Use](#). [Privacy Policy](#).

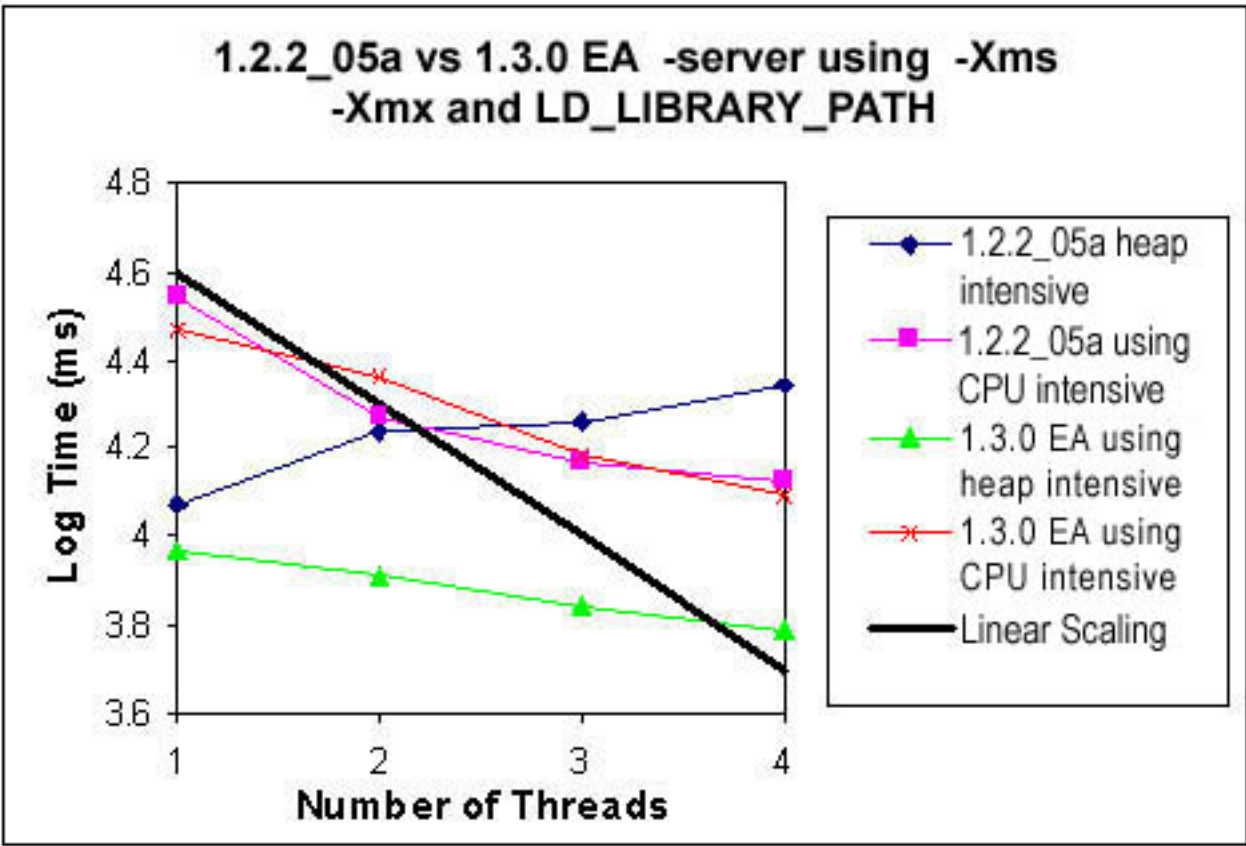
### Default Settings for 1.2.2\_05a vs Using -Xoptimize and -Xgenconfig



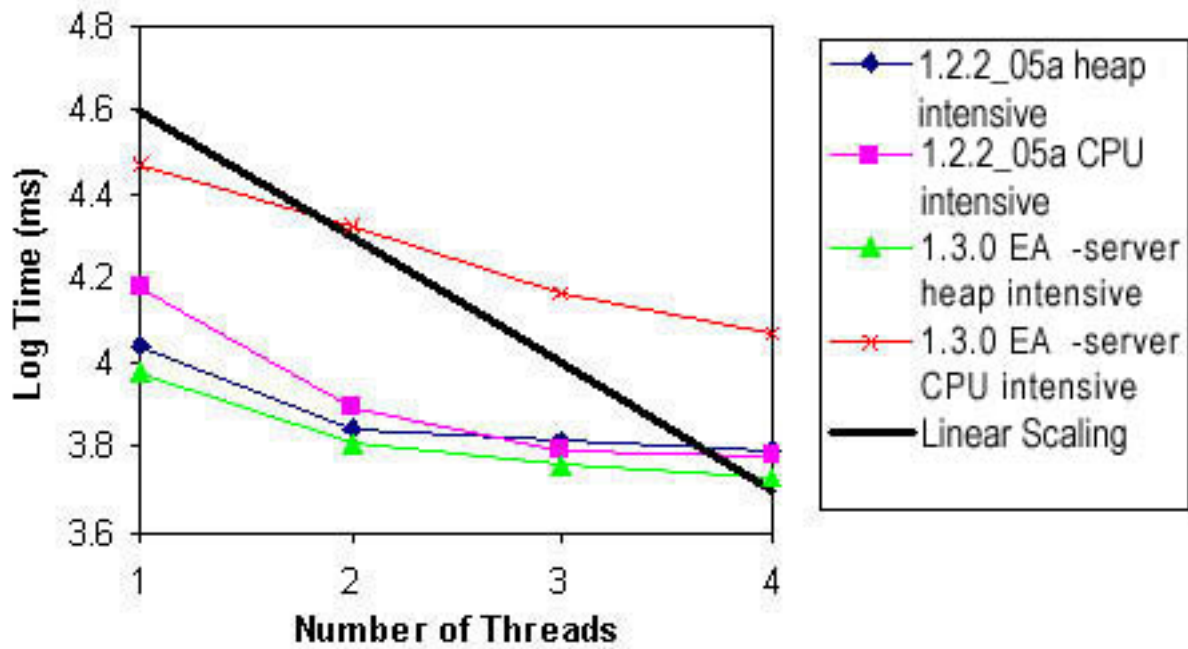


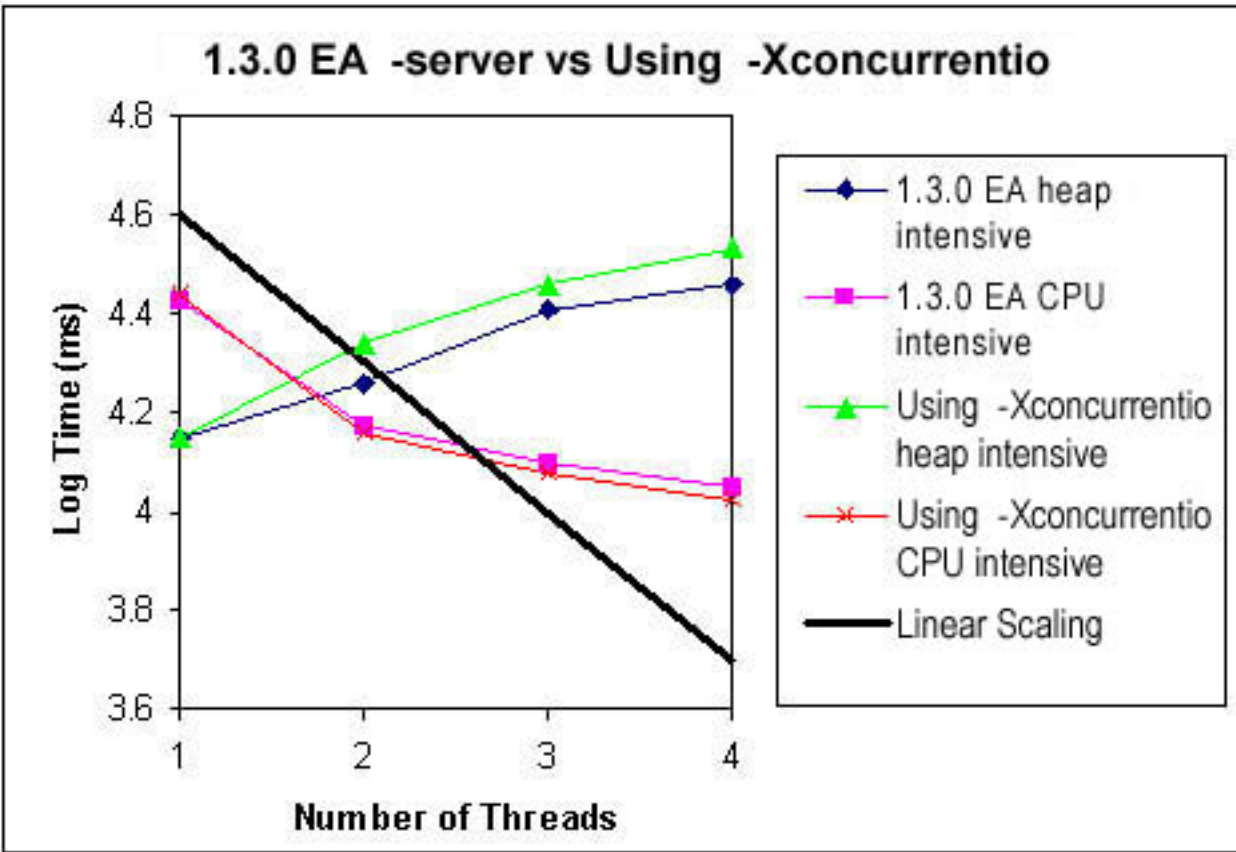






### 1.2.2\_05a Using -genconfig and -Xoptimize vs 1.3.0 EA -server using -SurvivorRatio with LD\_LIBRARY\_PATH=/usr/lib/lwp





```
*
* 00/08/01 @(#)HeapThread.java 1.3
*
* Copyright (c) 2000 Sun Microsystems, Inc. All Rights Reserved.
*
* Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
* modify and redistribute this software in source and binary code form,
* provided that i) this copyright notice and license appear on all copies of
* the software; and ii) Licensee does not utilize the software in a manner
* which is disparaging to Sun.
*/
```

```
package heaptest;

public class HeapThread extends Thread {

    private int m_id = 0;
    private int m_numThreads = 0;
    private int[] m_data = null;
    private int m_numDataPoints = 0;
    private int m_numIterations = 0;
    private int m_numNodesToAlloc = 0;
    private int m_heapCycles = 0;
    private int m_cpuCycles = 0;
    private Barrier m_goFlag = null;
    private Node m_firstNode = null;

    public HeapThread(int id, int numThreads, int[] data, int numDataPoints,
                     int numIterations, int numNodesToAlloc, int heapCycles,
                     int cpuCycles, Barrier goFlag)
    {
        m_id = id;
        m_numThreads = numThreads;
        m_data = data;
        m_numDataPoints = numDataPoints;
        m_numIterations = numIterations;
        m_numNodesToAlloc = numNodesToAlloc;
        m_heapCycles = heapCycles;
        m_cpuCycles = cpuCycles;
        m_goFlag = goFlag;
        m_firstNode = new Node();
    }

    public void run() {

        // Wait for all the threads are ready to go
        m_goFlag.waitForGo();

        for (int iter = m_id; iter < m_numIterations; iter += m_numThreads) {
            for (int heapIter = 0; heapIter < m_heapCycles; heapIter++) {
                doHeapBoundStuff(m_numNodesToAlloc);
            }
            for (int cpuIter = 0; cpuIter < m_cpuCycles; cpuIter++) {
                doCPUBoundStuff(m_data, m_numDataPoints);
            }
        }

        void doHeapBoundStuff(int numNodesToAlloc) {
            if (m_firstNode.m_next == null) {
                for (Node node = m_firstNode; numNodesToAlloc > 0; numNodesToAlloc--) {
```

```
        node.m_next = new Node();
        node = node.m_next;
    }
}
else {
    while (m_firstNode.m_next != null) {
        m_firstNode.m_next = m_firstNode.m_next.m_next;
    }
}
}

/*
double sqrtByNewtonsMethod(double x) {
    double lastEst = x;
    double est = 0;
    double epsilon = 1e-10;

    if (x == 0) est = 0;
    else {
        for (;;) {
            est = (lastEst*lastEst + x)/(2*lastEst);
            if (-epsilon < (est-lastEst) && (est - lastEst) < epsilon) {
                break;
            } else lastEst = est;
        }
    }
    return est;
}
*/

void doCPUBoundStuff(int[] data, int numDataPoints) {
    int i = 0;
    double avg = 0;
    double var = 0;

    data[0] = m_id;
    data[1] = m_id + 1;

    for (i = 2; i < numDataPoints; i++) {
        data[i] = data[i-1] + data[i-2];
    }

    for (avg = 0, i = 0; i < numDataPoints; i++) {
        avg += (double)numDataPoints;
    }
    avg /= (double) numDataPoints;

    for (var = 0, i = 0; i < numDataPoints; i++) {
        double diff = (double) data[i] - avg;
        var += diff*diff;
    }
    var /= (double) (numDataPoints - 1);
}
}
```

```
*
* 00/08/01 @(#)Barrier.java 1.3
*
* Copyright (c) 2000 Sun Microsystems, Inc. All Rights Reserved.
*
* Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
* modify and redistribute this software in source and binary code form,
* provided that i) this copyright notice and license appear on all copies of
* the software; and ii) Licensee does not utilize the software in a manner
* which is disparaging to Sun.
*/
```

```
package heaptest;

public class Barrier {

    private int m_numThreads = 0;
    private int m_currentCount = 0;

    public Barrier(int numThreads) {
        m_numThreads = numThreads;
    }

    public void waitForGo() {
        synchronized (this) {
            m_currentCount++;
            if (m_currentCount < m_numThreads) {
                try {
                    wait();
                } catch (InterruptedException exp) {
                    System.out.println("Barrier caught interrupted exception!");
                }
            }
            else notifyAll();
        }
    }
}
```

```
*
* 00/08/01 @(#)Node.java 1.3
*
* Copyright (c) 2000 Sun Microsystems, Inc. All Rights Reserved.
*
* Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
* modify and redistribute this software in source and binary code form,
* provided that i) this copyright notice and license appear on all copies of
* the software; and ii) Licensee does not utilize the software in a manner
* which is disparaging to Sun.
*/
```

```
package heaptest;

public class Node {
    public Node m_next = null;
    private char[] m_payload = null;;
    static char[] k_payload = {'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a',
                               'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a'};

    public Node() {
        m_payload = new char[64];
        System.arraycopy(k_payload, 0, m_payload, 0, m_payload.length);
    }
}
```

```
*
* 00/08/01 @(#)HeapTest.java 1.3
*
* Copyright (c) 2000 Sun Microsystems, Inc. All Rights Reserved.
*
* Sun grants you ("Licensee") a non-exclusive, royalty free, license to use,
* modify and redistribute this software in source and binary code form,
* provided that i) this copyright notice and license appear on all copies of
* the software; and ii) Licensee does not utilize the software in a manner
* which is disparaging to Sun.
*/

package heaptest;

import java.io.*;

public class HeapTest {

    public static void main(String[] args) {

        int k_numIterations = 500;
        int k_numNodesToAlloc = 3000;
        int k_numDataPoints = 50000;

        int maxNumThreads = 0;
        int numThreads = 0;
        int totalCycles = 0;
        int heapCycles = 0;
        int cpuCycles = 0;
        int i = 0;

        try {

            if (args.length < 2) usage();
            PrintStream logFile = null;
            try {
                logFile = new PrintStream(new FileOutputStream(args[2]));
            } catch (Exception e) {
                System.out.println("Unable to open log file. Printing to System.out...");
                logFile = System.out;
            }

            try {
                maxNumThreads = Integer.parseInt(args[0]);
                totalCycles = Integer.parseInt(args[1]);
            } catch (Exception e) {
            }

            if (maxNumThreads == 0 || totalCycles == 0) usage();
            else {
                logFile.println("\nMax # threads =      " +
                    maxNumThreads + "\n" +
                    "Total (heap + CPU) cycles = " +
                    totalCycles + "\n\n");
            }

            HeapThread[] threads = new HeapThread[maxNumThreads];
            logFile.print("# Threads");
            for (cpuCycles = 0, heapCycles = totalCycles; cpuCycles <= totalCycles;
                cpuCycles++, heapCycles--) {
                logFile.print("\t" + heapCycles + " Heap, " + cpuCycles + " CPU");
            }
        }
    }
}
```

```
    }
    logfile.flush();

    for (numThreads = 1; numThreads <= maxNumThreads; numThreads++) {
        logfile.print("\n\n" + numThreads);
        for (cpuCycles = 0, heapCycles = totalCycles; cpuCycles <= totalCycles;
cpuCycles++, heapCycles--) {

            Barrier goFlag = new Barrier(numThreads);
            for (i = 0; i < numThreads; i++) {
                threads[i] = new HeapThread(i,
                                            numThreads,
                                            new int[k_numDataPoints],
                                            k_numDataPoints,
                                            k_numIterations,
                                            k_numNodesToAlloc,
                                            heapCycles,
                                            cpuCycles,
                                            goFlag);
            }

            long elapsedTime = System.currentTimeMillis();
            for (i = 0; i < numThreads; i++) {
                threads[i].start();
            }

            for (i = 0; i < numThreads; i++) {
                threads[i].join();
            }
            elapsedTime = System.currentTimeMillis() - elapsedTime;
            logfile.print("\t" + elapsedTime);
        }
    }
    logfile.flush();
} catch (Exception e) {
    System.out.println("Caught exception!!");
    e.printStackTrace();
}

}

public static void usage() {
    System.out.println("java HeapTest <numThreads> <num of (CPU + Heap) cycles>");
    System.exit(1);
}
}
```