Efficiency of Thread-parallel Java Programs from Scientific Computing

Holger Blaar, Matthias Legeler, Thomas Rauber

Institut für Informatik, Universität Halle-Wittenberg, 06099 Halle (Saale), Germany
E-mail: {blaar,legeler,rauber}@informatik.uni-halle.de

Abstract

Many applications from scientific computing can benefit from object-oriented programming techniques because of their flexible and modular program development support. On the other hand, acceptable execution time can often only be reached by using a parallel machine. We investigate the support of Java for parallel execution. By using Java's thread mechanism we study how basic numerical methods can be expressed in parallel and which overhead is caused by thread management. We consider implementations that differ in scheduling, synchronization, and the management of threads. We obtain some guidelines to organize a multi-threaded Java program to get best performance.

1 Introduction

Many applications from scientific computing can benefit from object-oriented programming techniques, since they allow a flexible and modular program development. Often, these applications are computation-intensive, so the sequential execution time is quite large. Therefore it is profitable to use a parallel machine for the execution. But up to now, no parallel object-oriented programming language has been established as a standard.

Java is a popular language and has support for a parallel execution integrated into the language. Hence, it is interesting to investigate the usefulness of Java for executing scientific programs in parallel. To support such an investigation, we have performed several tests with basic numerical methods which we describe in this paper. In particular, we address the questions how the numerical computations can be expressed in parallel using the thread mechanism of Java, which overhead is caused by the thread management and which possibilities the programmer has to influence the parallel execution. We consider several implementations of the numerical methods which differ in thread management, thread synchronization, and thread scheduling and compare the resulting performance. As platform, we use a Sun E5500 and a E10000 system. The investigations result in some guidelines how the programmer should organize his multi-threaded Java program to get the best performance.

There are many projects that consider the usefulness of Java for high-performance scientific computing. Many of these activities are united in the Java Grande Forum (`www.javagrande.org`) which includes projects that are exploring the use of Java for high-performance scientific and numerical computing [1, 4] and groups that are developing suggestions for future Java extensions like extensions to the representation of floating-point numbers and complex numbers and multidimensional array representations. Other activities include projects to extend Java for high-performance parallel or distributed computing like the Titanium project [11] or the JavaParty project [9]. An MPI-like API for Java (MPJ) has been proposed in [3].

The rest of the paper is organized as follows: Section 2 describes several aspects of thread-parallel programming with Java and discusses how the programmer can influence the parallel execution. Section 3 considers the overhead of a multi-threaded execution of programs. Section 4 describes experiments with different versions of Java programs implementing basic numerical algorithms. Section 5 concludes the paper.

2 Java thread parallelism and scheduling

Thread creation: In contrast to most other programming languages where the operating system and a specific thread library like Pthreads [2] or C-Threads are responsible for the thread management, Java has a direct support for multithreading integrated in the language, see, e.g., [8]. The `java.lang` package contains a thread API consisting of the class `Thread` and the interface `Runnable`. There are two basic methods to create threads in Java.

Threads can be generated by specifying a new class which inherits from `Thread` and by overriding the `run()` method in the new class with the code that should be executed by the new thread. A new thread is then created by generating an object of the new class and calling its `start()` method.

An alternative way to generate threads is by using the interface `Runnable` which contains only the abstract method `run()`. The `Thread` class actually implements the `Runnable` interface and, thus, a class inheriting from `Thread` also implements the `Runnable` interface. The creation of a thread without inheriting from the `Thread` class consists of two steps: At first, a new class is specified which implements the `Runnable` interface and overrides the `run()` method with the code that should be executed by the thread. After that, an object of the new class is generated and is passed as an argument to the constructor method of the `Thread` class. The new thread is then started by calling the `start()` method of the `Thread` object. A thread is terminated if the last statement of the `run()` method has been executed. An alternative way is to call the `interrupt()` method of the corresponding `Thread` object. The method `setPriority(int prio)` of the `Thread` class can be used to assign a priority level between 1 and 10 to a thread where 10 is the highest priority level. The priority of a thread is used for the scheduling of the user level threads by the thread library, see below.

Thread synchronization: The threads of one program have a common address space. Thus, access to the same data structures have to be protected by a synchronization mechanism. Java supports synchronization by implicitly assigning a lock to each object. The easiest way to keep a method thread-safe is to declare it `synchronized`. A thread must obtain the lock of an object before it can execute any of its `synchronized` methods. If a thread has locked an object, no other thread can execute any other `synchronized` method of this object at the same time.

An alternative synchronization mechanism is provided by the `wait()` and `notify()` methods of the `Object` class which also supports a list of waiting threads. Since every object in the Java system inherits directly or indirectly from the `Object` class, it is also an `Object` and hence supports this mechanism. When a thread calls the `wait()` method of an object, it is added to the list of waiting threads for that object and stops running. When another thread calls the `notify()` method of the same object, one of the waiting threads is woken up and is allowed to continue running. These basic synchronization primitives can be used to realize more complex synchronization mechanisms like barriers, condition variables, semaphores, event synchronization, and monitors, see, e.g., [8].

Scheduling of Java threads: Thread creation and management are integrated into the Java language and runtime system and are thus independent of a specific platform. The mapping of the threads that are generated by a user program (user level threads) to the kernel threads of the operating system depends on the implementation of the Java Virtual Machine (JVM). The most flexible multithreading model is the many-to-many model which is, e.g., realized in SOLARIS Native Threads provided by the SOLARIS 2.x operating system. This model allows many threads in each user process. The threads of every process are mapped by the threads library to the kernel threads of the operating system which are then mapped to the different processors of the target machine by the scheduler of the operating system. The intention of this two-layer thread model is to decouple the scheduling and management of user threads from the kernel [7]. User threads have their own priority scheme and are scheduled with a separate scheduling thread that is automatically created by the threads library.

SOLARIS ***scheduling:*** The SOLARIS Native Threads library uses lightweight processes (LWPs) to establish the connection between the user threads and the kernel threads. The user threads of a process are mapped to the LWPs by the SOLARIS threads library. There is a pool of LWPs available for the mapping of user threads to kernel threads. If there are more running user threads in a process than available LWPs, user threads must wait either until new LWPs are added to the pool or until one of the running LWPs is freed. Any Java thread may run on any LWP. When a thread is ready to run, the user-level scheduler assigns an LWP from the pool to it. The LWP will continue to run the thread until either a thread at a higher priority becomes runnable or the thread blocks on a synchronization.

Each LWP is mapped to a kernel thread which are mapped by the scheduler of the operating system to the processors. The number of LWPs is controlled by the threads library of SOLARIS 2.x and is automatically adapted to the number of user threads generated by the program. One way to influence the number of LWPs directly is by using the function `thr_setconcurrency (int n)` to add n new LWPs to the pool of available LWPs. This function is provided by the SOLARIS threads library which can be accessed via the Java Native Interface (JNI). A new LWP will execute a specific function in the threads library that looks for runnable threads in the library's queue. If none exists, the LWP blocks on a timed condition variable. If the timer expires (which takes five minutes by default) and the LWP has not executed a user thread, the LWP will terminate [7]. The scheduling of the user threads by the threads library can be influenced indirectly by setting the priorities of the user threads, (see above): a thread at a given priority level does not run unless there are no higher-priority threads waiting to run.

Solaris uses scheduling classes and priorities for the scheduling of the kernel threads (and their associated LWPs). The following classes are supported: timesharing (priorities 0–59), interactive (0–59), system (60–99) and real-time (100–159). Higher values mean higher priorities. Usually, LWPs are generated with timeshare priority. The exact priorities of the LWPs are set by SOLARIS 2.x. The programmer can give hints to raise or lower the priority of a process (affecting all its kernel threads) using the `priocntl()` command. It can be observed that processes with a large number of LWPs get a higher priority than other processes.

The difficulty of experiments with Java threads lies in the fact that the scheduling is not under the control of the programmer. When starting a thread-parallel program, it is not clear which thread will be assigned to which LWP and which LWP will be executed on which processor. The number of LWPs can be set to a specific value, but the operating system can adapt this number during the execution of the program. The number of processors used for the execution of a program is completely under the control of the operating system and can vary during the execution of a thread-parallel Java program.

3 Runtime of multi-threaded programs

For measuring the runtime of a parallel application, `java.lang.System` provides a method `currentTimeMillis()` to obtain the *real time* (wall clock time) of an application in milliseconds, so the time measurement can be influenced by other applications running on the system. The user and system CPU time of an application can be obtained by using the C function `times()` from `<sys/times.h>` via the JNI. For a parallel application, this provides the *accumulated* user and system CPU time UCT$_p$ on all processors, i.e., to compute the actual parallel CPU time, these values have to be divided by the number of processors used. As explained above, the number of processors used for a parallel application cannot easily be determined and can also vary during the execution of a program. Information on the number P$_A$ of processors used for an application can be obtained with the Unix `top` command, but this only works with a significant time delay and does not provide reliable information for parallel executions with a small execution time. Thus, the following can be concluded:

- The *sequential* user and system CPU time UCT$_s$ can be obtained accurately with the `times()` function.

- The real time RT can be obtained quite accurately with `currentTimeMillis()`, thus we can compute a speedup $S_{RT} = \text{UCT}_s/\text{RT}$ which provides good results on an unloaded system and can be considered as a lower bound on the speedup that has really been achieved by a specific program run. We use these speedups in the following, since it is the most reliable information, even if we have not always been able to use an unloaded system.

Table 1. Thread overhead in milliseconds.

ULT	#LWPs = 10 $\frac{UCT}{ULT}$	#LWPs = 10 $\frac{RT}{ULT}$	#LWPs = 100 $\frac{UCT}{ULT}$	#LWPs = 100 $\frac{RT}{ULT}$	#LWPs = 1000 $\frac{UCT}{ULT}$	#LWPs = 1000 $\frac{RT}{ULT}$
1000	0.37	0.47	0.39	0.59	0.47	1.00
5000	0.30	0.44	0.35	0.43	0.29	0.82
10000	0.30	0.42	0.32	0.44	0.31	0.75

- Using UCT$_p$, a speedup value can be computed by $S_c = \text{UCT}_s \cdot P_A/\text{UCT}_p$, but this speedup value can be quite unreliable, if the value of P_A is unreliable. In the following, we give P_A for most applications, but we must point out that the value might be too small because of the time delay of the `top` command.

Overhead of thread execution: The creation and management of Java threads is completely under the control of the threads library and does not involve the operating system. To get an impression of the overhead of creating, managing, and deleting a thread, we have performed runtime tests on a SUN-Enterprise E5500 with 12 336 MHz UltraSPARC processors. Table 1 shows the time (user CPU time UCT) for the execution of a specific number of user level threads (ULT) executing an empty `run()` method with 10 dummy parameters. The tests have been performed with JDK 1.2.1 and SOLARIS 2.6 (Sun OS 5.6).

For the real-time measurements, we tried to use an unloaded system to avoid the influence of other processes running on the system. The following observations can be made:

- The user CPU time UCT for thread overhead, which also includes the scheduling of the user threads, usually lies between 0.3 ms and 0.4 ms, independently from the number of user threads and LWPs.

- The real time for thread overhead, which also takes the time for the scheduling of the LWPs into account, increases with the number of LWPs.

This indicates that the scheduling of the user threads does not cause a significant overhead whereas the scheduling of the LWPs causes runtime costs that increase considerably with the number of LWPs, since the operating system is involved. This suggests to adjust the number of LWPs to the number of processors available.

4 Thread-parallel execution of numerical algorithms in Java

In this section, we give a short description of thread-parallel implementations of several standard numerical computations in Java. The implementations are based on basic classes like `Vector` or `Matrix` containing the underlying data structures and initializations. The numerical methods are realized as `run()`-methods of separate classes which inherit from the base classes. The runtime experiments have been performed on a SUN Enterprise E5500 with 12 336 MHz UltraSPARC processors, JDK 1.2.1 and SOLARIS 2.6 (Sun OS 5.6). The machine was not unloaded, so jobs of other users were running and could influence the time measurements.

4.1 Matrix multiplication

Table 2 shows the resulting runtime in seconds and speedup values for the multiplication of two matrices of size 1000×1000. ULT denotes the number of user level threads used for the execution. LWP is the number of LWPs generated by calling `thr_setconcurrency()` or automatically assigned by the thread library. P_R denotes the number of requested processors by the user process. P_A is the average number of processors assigned to the executing user process measured by `top`. RT discribes the real time in seconds, and S_{RT} denotes the speedup computed by S_{RT} = UCT$_s$/RT where UCT$_s$ is the user CPU time of the corresponding sequential program in Java without threads. The execution of the computations is distributed among a specific number of threads where each thread is responsible for the computation of a row block of the result matrix, i.e., a row-blockwise distribution of the computations of the rows of the result matrix among the threads is used.

Table 2. Thread-parallel execution of a matrix multiplication.

ULT	LWP	P_R	P_A	RT	S_{RT}
1	1	1	1.0	381	0.9
4	4	4	3.9	95	3.8
8	8	8	6.8	54	6.6
12	12	12	7.8	42	8.5
16	16	12	8.5	38	9.4
26	26	12	8.5	34	10.5

The results in Table 2 and further runtime tests allow the following observations:

- If the number of LWPs is not explicitly set by the Java program, the thread library creates for each Java thread an LWP, until a fixed upper bound is reached (26 on the Sun E5500).
- The speedup is increasing with the number of LWPs. This effect can be observed, even if the number of LWPs is getting larger than the number of processors of the parallel system, i.e., even if there is an additional overhead for the scheduling of the LWPs by the scheduler of the operating system. This effect can be explained by the fact that processes with a larger number of LWPs get a larger priority and can therefore benefit from the priority-based scheduling of the operating system.
- If the number of user threads is smaller than the number of LWPs requested by `thr_setconcurrency()`, the thread library only creates one LWP for each user thread.

4.2 Gaussian Elimination

Gaussian elimination is a popular method for the direct solution of linear equation systems $Ax = b$ [5]. For a system of size n, the method performs n steps to convert the system $Ax = b$ to a system $Ux = y$ with an upper triangular matrix U. Each step k performs a division operation which divides row k by its diagonal element followed by $n - k$ elimination operations modifying rows $k + 1, \dots n$ with row k. The elimination operations can only be started after the preceding division operation has been completed. Hence, a synchronization has to be used between the division operation and the elimination operations of step k. Similarly, step $k + 1$ can only be started after the elimination operations of the preceding step have been completed, making another synchronization necessary. After the generation of the upper triangular system $Ux = y$, the solution vector x is determined by a backward substitution.

We have implemented several Java versions of this algorithm with pivoting differing in the specific thread creation and the synchronization of the threads [6].

Synchronization by thread creation: Step k, $1 \leq k < n$ is parallelized by generating a number of threads and assigning each thread a contiguous block of rows of the elimination submatrix. For the division operation, a small number of threads is used in a similar way. At the end of the division operation and the elimination phase of step k, the created threads are terminated and synchronized by a `join()` operation, so the necessary synchronizations are performed by thread termination. Thus, it can be guaranteed that the operations of the previous elimination step are completed before the next elimination step starts.

Figure 1 shows the corresponding program fragment. Ab denotes the Matrix A with the right side vector b of the linear system to solve. In the run method of the class `EliminationThread` the elimination steps are carried out. Table 3 shows some results of our tests with a linear system of size $n = 1500$. The same notation as in Table 2 is used.

```
Thread thE[] = new Thread[ne];
// array for threads (elimination step)
//...
for (int k=0; k<n; k++) {
  // division step
  // ...
  A[k,k] = 1;
  for (int i=k+1; i<ne; i++) {
    thE[i] = new EliminationThread(Ab,k,i);
    thE[i].setPriority(Thread.MAX_PRIORITY);
    thE[i].start(); } //starts the run method
          // of the elimination thread class
  for (int i=k+1; i<ne; i++) {
    try {thE[i].join();} // synchronization
      catch (...) {} }
}
```

Figure 1. Gaussian elimination with synchronization by thread creation (code fragment).

The division step has not been parallelized, since a parallelization reduces the attained speedups.

Table 3. Gaussian elimination with synchronization by thread creation.

ULT	P_R	P_A	RT	S_{RT}
1	1	1.0	187	0.7
4	4	3.1	59	2.4
8	8	4.4	42	3.3
12	12	4.7	42	3.3
16	12	5.3	38	3.7
20	12	5.3	38	3.7
40	12	5.2	47	3.0

The execution times and speedups show that synchronization by thread creation only leads to a limited speedup, since the overhead for thread creation and waiting time in each step is quite large compared to the computational work of each step.

Synchronization by active waiting: For this version, the threads are created at the beginning of the modification of $Ax = b$ and each thread is assigned to a number of rows whose computations it has to perform. The assignment of rows to threads can be performed blockwise or cyclically. For n rows and t threads, the blockwise scheme assigns rows $s \cdot n/t, \ldots, (s+1) \cdot n/t - 1$ to thread number s, if we assume that n is a multiple of t. For the cyclic scheme, row i is assigned to thread $i \bmod t$. For the blockwise assignment, a thread is terminated as soon as the rows assigned to it are modified, i.e., the number of ULTs is decreasing during the computation. For the cyclic assignment, the number of ULTs is constant during most of the computation. Only during the modification of the last t rows the number of ULTs is decreasing.

The synchronization between the division operations and the elimination operations of a step k is arranged by using a bitvector `bv` with a flag for each row indicating whether the division operation for this row has been completed (`true`) or not (`false`). The threads executing the corresponding division operation set the flag after the completion of the operation. Before starting the elimination operations of step k, all participating threads read `bv[k]` iteratively until it has the value `true`. Table 4 shows some tests with an equation system of size $n = 1500$ with the same notation as in Table 3.

For up to 8 ULTs, a cyclic assignment of rows to threads leads to slightly better speedup values than a blockwise assignment, mainly because of a better load balance. For a larger number of ULTs, a blockwise assignment of the rows to the ULTs leads to better speedup values, mainly because of smaller waiting times at synchronization points: The thread performing the division operation of a row also performs the division operations of the following rows and has to perform less elimination operations than the remaining threads. Therefore, this thread will complete the following division operation usually before the other threads have completed their elimination operations. In this case, no waiting time is required at the synchronization points. For a cyclic assignment, each thread has to perform nearly the same number of elimination operations. The division step of neighboring rows are performed by different threads. Thus, it can easily happen that a thread has completed its elimination operations before the thread that is responsible for the division operation has completed this operation. The resulting waiting times may reduce the attained speedups considerably. On the other hand, there are more ULTs than processors; so after the termination of a ULT T, another ULT can be assigned to the corresponding LWP and can thus run on the processor that has executed T before. Hence, load balancing does not play a dominant role.

Table 4. Gaussian elimination with synchronization by active waiting.

ULT	P_R	cyclic			blockwise		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	1.0	150	0.9	1.0	150	0.9
4	4	3.4	36	3.9	2.5	49	2.8
8	8	5.4	24	5.8	3.6	26	5.3
12	12	8.0	39	3.6	4.0	17	8.2
16	12				4.4	16	8.7
20	12				4.4	14	9.9
40	12				5.8	16	8.7

Barrier synchronization: The synchronization is achieved with an object `barr` of the `Barrier` class. The barrier object is initialized for the number of participating threads. At the synchronization points required, each thread enters the barrier by calling `barr.waitForRest(int n)`. The last thread entering causes the release of the barrier and all waiting threads can continue execution. Some runtime results are shown in Table 5 (problem size $n = 1500$). The small speedup values can be explained by the fact that each thread has to wait for all other threads at the synchronization points, not only for the one thread performing the division operation. Thus, the delay of one of the threads can easily cause all other threads to wait.

Table 5. Gaussian elimination with barrier synchronization.

ULT	P_R	cyclic			blockwise		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	1.0	150	0.9	1.0	151	0.9
4	4	3.1	42	3.3	2.3	54	2.6
8	8	3.1	40	3.5	2.8	42	3.3
12	12	3.4	36	3.9	3.1	37	3.8
16	12	3.6	33	4.2	3.4	34	4.1
20	12	3.6	34	4.1	3.4	33	4.2
40	12	4.0	40	3.5	3.6	33	4.2

Event semaphore: The synchronization is achieved by creating an object of the `EventSemaphore` class for each row of the equation system. Before starting an elimination step, each thread waits for the completion of the preceding division step by calling a method `WaitForEvent()`. The thread that performs this division step sets the event by calling `SetEvent()`, thus waking up all waiting threads.

Table 6. Gaussian elimination with synchronization by event semaphores.

ULT	P_R	cyclic			blockwise		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	1.0	152	0.9	1.0	152	0.9
4	4	3.0	40	3.5	2.5	51	2.7
8	8	3.5	33	4.2	3.4	25	5.6
12	12	3.8	26	5.3	3.8	19	7.3
16	12	4.2	24	5.8	4.2	17	8.2
20	12	4.2	23	6.0	4.3	15	9.3
50	12	4.1	30	4.6	4.7	12	11.6

A corresponding program fragment is shown in Figure

```
EventSemaphore sVec[]=
              new EventSemaphore[Ab.row];
for (int i=0; i<Ab.row; i++)
  sVec[i] = new EventSemaphore(false);
...
Thread th[] = new Thread[nth];
CPUsupport.setConcurrency(lwp);
for (int i=0; i<nth; i++) {
  th[i] = ElimEvSem(Ab, i, range, sVec);
  th[i].setPriority(Thread.MAX_PRIORITY);
  th[i].start();
}
for (int i=0; i<nth; i++) {
  try {th[i].join();}
  catch (...) {}
}
...
// run method
public void run() {
  for (int i=0; i<range[curr_th][1]; i++) {
    if (i!=0) {
      if (i<range[curr_th][0]+1)
                sVec[i-1].waitForEvent();
      ...    // perform elimination step
    }
  }
  if (i>range[curr_th][0]-1) {
    ...      // perform division step
    sVec[i].setEvent();
  }
}
```

Figure 2. Gaussian elimination with synchronization by event semaphores (code fragment).

2. The blocks of the rows of A are accessible by the `range` array using the corresponding thread with the current thread number `curr_th`.

Table 6 presents some runtime results with linear systems of size $n = 1500$. The synchronization by event semaphores has similar properties as the synchronization by active waiting. In particular, the threads executing the modification operation are only waiting for the one thread that executes the division operation and not for all threads as it was the case for barrier synchronization. Synchronization by event semaphores leads to slightly better speedup values as synchronization by active waiting especially for a larger number of ULTs, since a waiting thread can be suspended while waiting for the event to occur.

4.3 Jacobi and Gauß-Seidel Relaxation

Jacobi and Gauß-Seidel relaxation are iterative methods for the solution of a linear equation system $Ax = b$. The solution of this system can be approximated by iteratively computing approximation vectors $x^{(k)}$ according to

$$Mx^{(k+1)} = Nx^{(k)} + b$$

for $k = 1, 2, \ldots$. M and N are appropriate matrices that are computed from A [5]: Let L be the lower triangular submatrix of A, U be the upper triangular submatrix of A and D be the diagonal matrix that contains the diagonal entries of A, so it is $A = U + L + D$. Jacobi and Gauß-Seidel iterations differ in the definition of M and N. For Jacobi, $M = -D$, $N = -(L + U)$ is used, for Gauß-Seidel $M = -(D + L)$ and $N = -U$. Thus, Jacobi uses for the computation of each component of $x^{(k+1)}$ the previous approximation vector whereas Gauß-Seidel uses the most recent information, i.e., for the computation of $x_i^{(k+1)}$ all components $x_j^{(k+1)}$ for $j < i$, which have previously been computed in the same approximation step are used.

For this reason, Jacobi relaxation can be easily parallelized for distributed memory machines (DMMs), because data exchange needs to be performed only at the end of each approximation step. On the other hand, Gauß-Seidel relaxation is considered to be difficult to be parallelized for DMMs, since a data exchange has to be performed after the computation of each component of the approximation vector. At the end of each time step of a Jacobi or Gauß-Seidel iteration, all participating threads need to be synchronized, so that the next iteration is executed with the current values of the iteration vector. Thus, a barrier synchronization is appropriate for separating the time steps. For the Gauß-Seidel iteration, additional synchronizations need to be performed within the time steps to ensure that always the newest values of the components of the iteration vectors are used. Here, barrier synchronization, active waiting or event semaphores can be used.

Tables 7 – 9 show the resulting performance characteristics. As example, a system of $n = 4000$ equations has been used. The parallel implementations differ in the synchronization mechanisms used: synchronization by thread creation and barrier synchronization for the synchronization between time steps, and barrier synchronization, active waiting and event semaphore for the synchronization within time steps for Gauß-Seidel iteration. Not all are shown in the Tables.

Table 7. Jacobi iteration with synchronization by thread creation and barriers.

ULT	P_R	thread creation			barrier		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	1.0	261	0.9	1.0	236	1.0
2	2	1.9	132	1.8	1.9	119	2.0
4	4	3.4	75	3.2	3.2	76	3.2
8	8	4.0	72	3.3	3.1	85	2.8
12	12	4.9	62	3.9	2.9	100	2.4
16	12	4.8	66	3.7	2.9	108	2.2

Table 8. Gauß-Seidel iteration with synchronization by thread creation and barriers, active waiting within iteration steps.

ULT	P_R	thread creation			barrier		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	0.9	58	0.9	1.0	58	0.9
2	2	1.6	30	1.8	1.6	31	1.8
4	4	2.5	20	2.7	2.5	21	2.6
7	7	4.6	36	1.5	5.2	70	0.8

Table 9. Gauß-Seidel iteration with synchronization by thread creation and barriers, loose synchronization within iteration steps

ULT	P_R	thread creation			barrier		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	1.0	54	1.0	1.0	54	1.0
2	2	2.0	28	2.0	2.0	27	2.0
4	4	3.4	16	3.4	3.8	14	3.9
8	8	5.7	10	5.5	6.3	9	6.1
12	12	6.9	8	6.8	6.5	8	6.8
16	12	7.2	7	7.8	7.2	8	6.8

Tables 7 and 8 show that for up to 4 threads, all variants lead to good speedup values. For 4 threads, the speedup values for the Jacobi iteration are slightly higher than for the Gauß-Seidel iteration, mainly because of the additional synchronization operations within the time steps for the Gauß-Seidel iteration.

For a larger number of threads, the speedup values are only increasing slightly (Jacobi) or are even dropping (Gauß-Seidel). This can be explained by the large synchronization overhead of thread creation or barrier synchronization that has already been observed for Gaussian elimination.

Loose synchronization: Moreover, we have implemented a variant with a loose synchronization method that synchronizes the executing threads only for the convergence test of the iteration, but not for the swapping from the old to the new iteration vector. For a Jacobi relaxation, this may result in a thread accessing a part of the iteration vector that has already been updated by another thread instead of accessing the entries of the previous iteration. This behavior does not affect the correctness of the resulting approximation and may even lead to a faster convergence. Table 10 shows that for a larger number of threads, the variant with the loose synchronization yields satisfactory speedups.

Table 10. Jacobi and Gauß-Seidel iteration with loose synchronization between and within iteration steps.

ULT	P_R	Jacobi			Gauß-Seidel		
		P_A	RT	S_{RT}	P_A	RT	S_{RT}
1	1	1.0	54	1.0	1.0	54	1.0
2	2	2.0	27	2.0	2.0	27	2.0
4	4	3.7	15	3.7	3.8	14	3.9
8	8	5.4	10	5.5	5.9	9	6.1
16	12	7.0	7	7.8	6.6	7	7.8
40	12	7.7	6	9.1	7.1	6	9.1

General observations: Considering the relation $R = \#ULT/\#LWP$ between the number of ULTs and LWPs, the following can be observed for all numerical methods considered: up to a specific relation R_{max}, the speedups are increasing with the value of R, but starting with R_{max}, increasing values of R lead to decreasing speedups. The reason for this effect lies in the fact that up to R_{max}, the decrease of the execution time because of a better parallelization outperforms the increase in the overhead for thread management, but starting with R_{max}, this effect reverses. The specific value of R_{max} depends on the application considered and the specific synchronization mechanism used.

5 Conclusions

The experiments described in this article show that both the synchronization method and the assignment of computations to threads play an important role for obtaining good speedups for basic numerical methods. Both the mapping and synchronization have to be chosen carefully and have to fit together. Moreover, the experiments with the Gaussian elimination show the interesting fact that for thread-based computations it might be beneficial to use a completely different mapping of computations to threads than for other programming models. For a distributed address space, a double-cyclic mapping of rows and columns to processors usually leads to the best speedups [10], whereas a blockwise distribution is not competitive because of a bad load balancing. For the thread-parallel execution in Java, the opposite behavior could be observed. The reason for this lies in the fact that the assignment of threads to processors is more flexible than the execution of message-passing processes on processors and that the synchronization of the threads plays an important role for the resulting execution time.

Acknowledgement

We thank the University of Cologne for providing access to their E5500 and E10000 systems.

References

[1] R.F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and Numerical Computing. *IEEE Computing in Science and Engineering*, 3(2):18–24, 2001.

[2] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman, Inc., 1997.

[3] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[4] G.C. Fox and W. Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience*, 9(6):415–425, 1997.

[5] G. Golub and C. Van Loan. *Matrix Computations*. John Hopkins University Press, 1989.

[6] M. Legeler. Untersuchungen zur Implementierung Thread-paralleler Algorithmen auf Shared-Memory-Rechnern. Master's thesis, Martin-Luther-Universität Halle-Wittenberg, 2000.

[7] J. Mauro and R. McDougall. *Solaris internals: core kernel components*. Sun Microsystems Press, 2001.

[8] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, Inc., 1999.

[9] M. Philippsen and M. Zenger. JavaParty – Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[10] T. Rauber and G. Rünger. Deriving Array Distributions by Optimization Techniques. *Journal of Supercomputing*, 15:271–293, 2000.

[11] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford*, 1998.