Computer Architecture Technical Report

TUCZ / RA-TR-96-05

Message Passing Efficiency on Shared Memory Architectures

Thomas Radke

RECHNERARCHITEKTUR

University of Technology Chemnitz–Zwickau

Department of Computer Science

Computer Architecture

Prof. Dr. W.Rehm

1 Introduction

Especially in the framework of the newly founded "Sonderforschungsbereich SFB 393 : Numerische Simulation auf massiv parallelen Rechnern", other parallel architectures – namely shared memory systems – are to be investigated for their ability to efficiently solve the addressed problems. Two principle ways are possible to implement parallel programs on shared memory architectures. First, one can use multithreading as the basic programming model for his/her implementation[1]. This guarantees the best use of the facilities of the underlying hardware. The other way is to use a message passing interface, that resides on top of multithreading, and write a CSP program.

The first way was evaluated at our department by porting a module of an existing message passing program (FEM solver) onto a shared memory system[2]. Tests showed that good efficiency results can be obtained for small problem sizes mapped onto few processors. But the port took a considerable amount of time, and writing a multithreaded program appeared to be even harder than defining the same problem with the message passing paradigm.

As there are comprehensive experiences at the University of Technology Chemnitz with message passing programming on the Parsytec massively parallel computer series, we concluded that the second approach should be favourized in the SFB 393.

This paper describes the potential facilities for message passing on 2 shared memory systems available at the Department of Computer Science: the KSR1 from Kendall Sqare Research, and Compaq's ProLiant 4000. The theoretical efficiencies of a message passing interface are given as communication bandwidth, communication latency, parallelity of communication and computation, and the implementation of global operations. Efficieny losses due to the implementation of the underlying multithreading functionality are not studied here in detail, the main intention was to evaluate the shared memory hardware architecture.

2 The KSR1 Architecture

The KSR1 parallel shared memory computer (built by Kendall Square Research) consists of 8 processor nodes. Each node has a KSR1 processor (20 MHz clock, RISC architecture, developed at Kendall Square Research) with a first-level cache of 0.5 MBytes, half for instructions, half for data. The node's main memory is 32 MBytes of size. It is called the local cache, but the local processor has access to local caches of other nodes too. This is realized by the KSR ALLCACHE architecture: all nodes are chained in a ring (called ALLCACHE engine). Any access to remote caches is done over this ring, with a physical bandwidth of 1 GByte/s. The coherency of global memory is ensured by a local cache directory in every node.

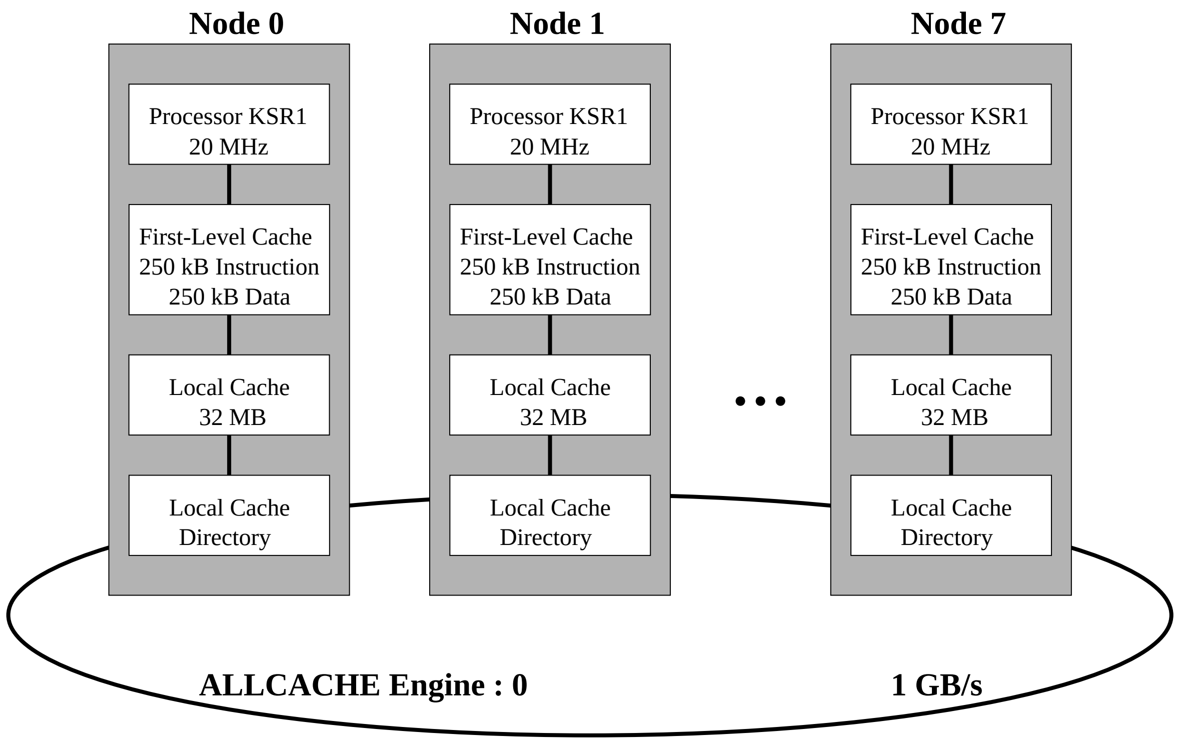


Figure 1: Architecture of the KSR1 computer [3]

Because there is no explicite global main memory, the KSR1 computer is classified as COMA architecture (cache only memory access).

3 The Compaq ProLiant 4000 Architecture

The ProLiant 4000 is a symmetric multiprocessor PC system, built by Compaq. It has 4 processors (Intel Pentium 66 MHz, 8 KBytes on-chip cache for instructions and data respecticely). Each processor module is supplied with its own second-level cache (256 KBytes). Access to main memory is realized by the Compaq TriFlex bus, with a peak bandwidth of 267 MBytes/s.

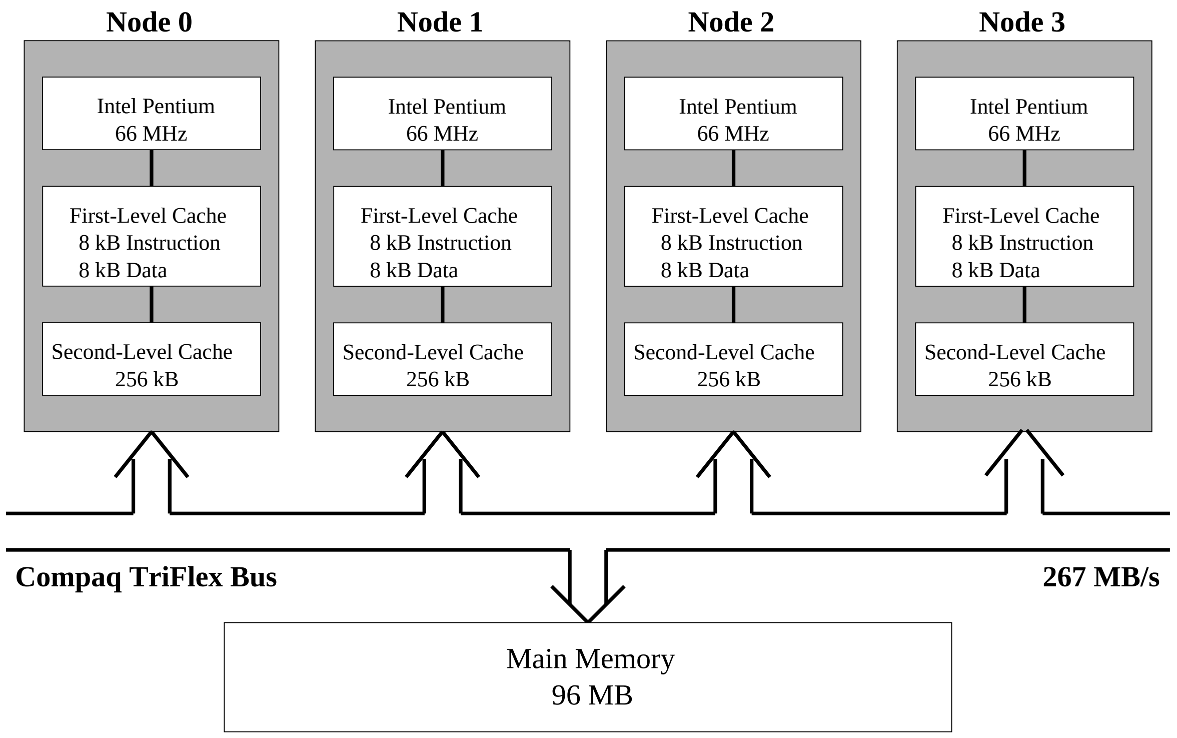


Figure 2: Architecture of the Compaq ProLiant 4000

The Compaq ProLiant 4000 has a UMA architecture (uniform memory access). All processors have the same access time to main memory.

4 The Multithreading Programming Model

Multithreading extends the process concept of classical UNIX operating systems by the opportunity, that a process itself can spawn several threads of control at runtime. The threads have an own stack and register set, so that they can run in parallel. The process address space is shared between all threads. Global variables can be used for inter-thread-communication. Synchronization facilities (mutexes, semaphores, condition variables, barriers) are used to implement controlled access to shared variables and to synchronize threads at specific program points.

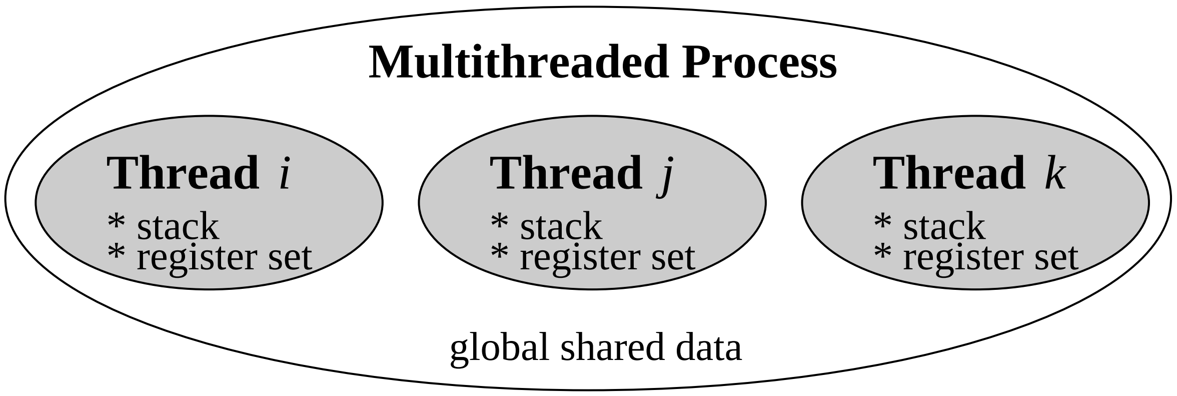


Figure 3: Threads in a multithreaded process

5 Message Passing on Top of Multithreading

The emulation of message passing in a multithreaded environment can be expressed as following:

- Threads become the nodes of the message passing program.
- Communication is done via synchronization of threads and simply copying of the data from the sending node to the receiver.

5.1 Threads as Nodes

In distinction to message passing programming under PARIX, where the whole main program is duplicated onto all processor nodes at load time, threads begin their execution in a start function from within the same process. This could be done implicitely by calling a function

```
    int MP_Init (unsigned int num_nodes, void (*start_fn) (void *arg));
```

with *start_fn* specifying the function to start execution of the thread with the single argument *arg*, and *num_nodes* determining the number of nodes to create.

Another distinction is the declaration of global variables. In a PARIX program, all global variables are visible to the local node only. In multithreaded programs these variables are shared between all nodes. A possible way to solve this problem is the introduction of private data. Each variable with an attribute *private* becomes thread-specific, i.e., every thread gets a local copy of the variable. This solution requires the explicite declaration of those variables by the application programmer and some changes in the compiler to support the handling with thread-specific data.

5.2 Node-to-Node Communication

Communication between nodes is very easy to implement because of the common address space of all threads. A (synchronous) communication operation requires the synchronization of the participating nodes and a data transfer operation (a simple memcpy) from the sender to the receiver(s).

Prototypes for a basic `SendMessage() / RecvMessage()` function are introduced in the following:

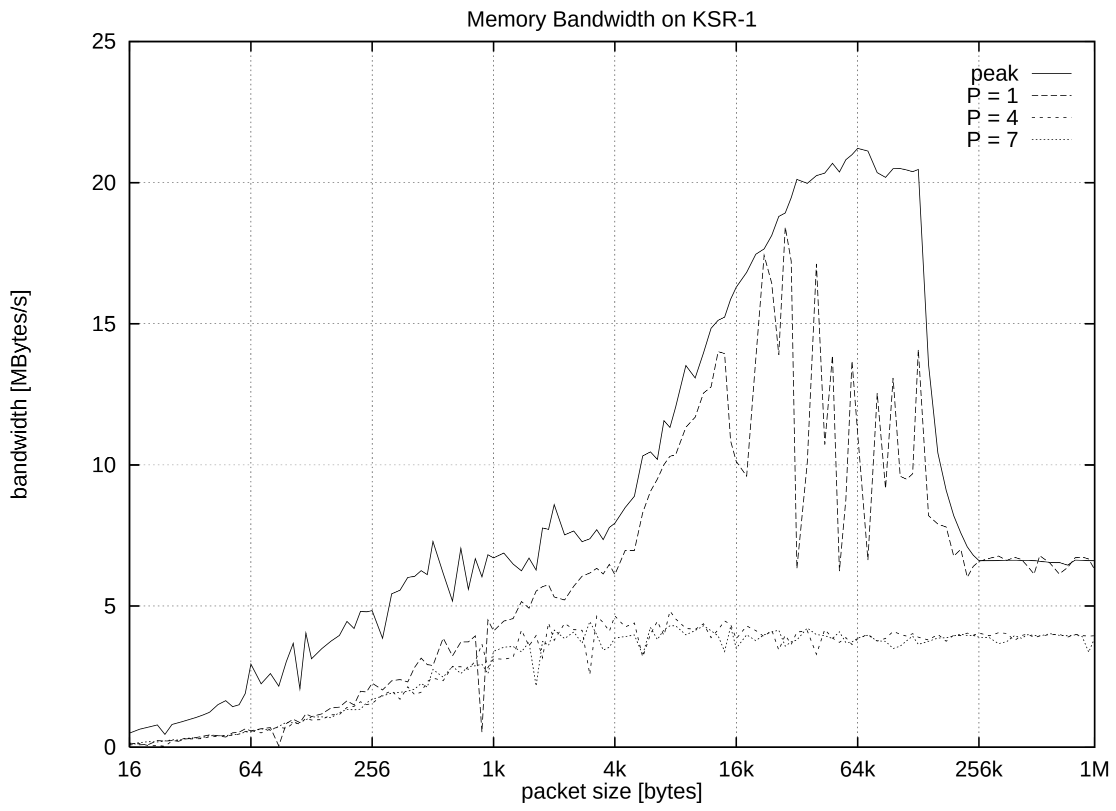
```
typedef struct {
    sem_t sendReady, recvReady;
    void *message;
    unsigned int bytes;
    unsigned int *transfered;
} Channel_t;

unsigned int SendMessage (void *message, unsigned int bytes, Channel_t *channel)
{
    unsigned int transfered;

    channel->bytes = bytes;
    channel->message = message;
    channel->transfered = &transfered;
    sem_signal (&channel->sendReady);
    sem_wait (&channel->recvReady);
    return (transfered);
}

unsigned int RecvMessage (void *message, unsigned int maxBytes, Channel_t *channel)
{
    unsigned int transfered;

    sem_wait (&channel->sendReady);
    transfered = *channel->transfered = min (maxBytes, channel->bytes);
    memcpy (message, channel->message, transfered);
    sem_signal (&channel->recvReady);
    return (transfered);
}
```

A *channel_t* structure is used for synchronous uni-directional communication. It contains a semaphore for the synchronization of the sending and the receiving node respectively, a pointer to the message to be passed, the size of the message in bytes, and a help variable the store the actual transfered number of bytes.

The `SendMessage()` function prepares the channel structure (sets the pointer to the message and the size), signalles the receiver that it is ready to send, and waits for the receiver to fulfil the operation. The receiver waits until the sender is ready to send, and then copies the message into its own buffer. It sets the actual length of the transfered message (the minimum of the requested sizes of the sender and the receiver) and signalles the sender for completion.

The time needed to synchronize the sender and the receiver depends on the implementation of the semaphore functions. If it is neglected, only the time of the memcpy operation remains. So the memory bandwidth of the underlying hardware architecture determines the peak communication bandwidth for a message passing emulation.

This was examined on both the KSR1 and the ProLiant 4000, with the packet size and the number of active processors as parameters:



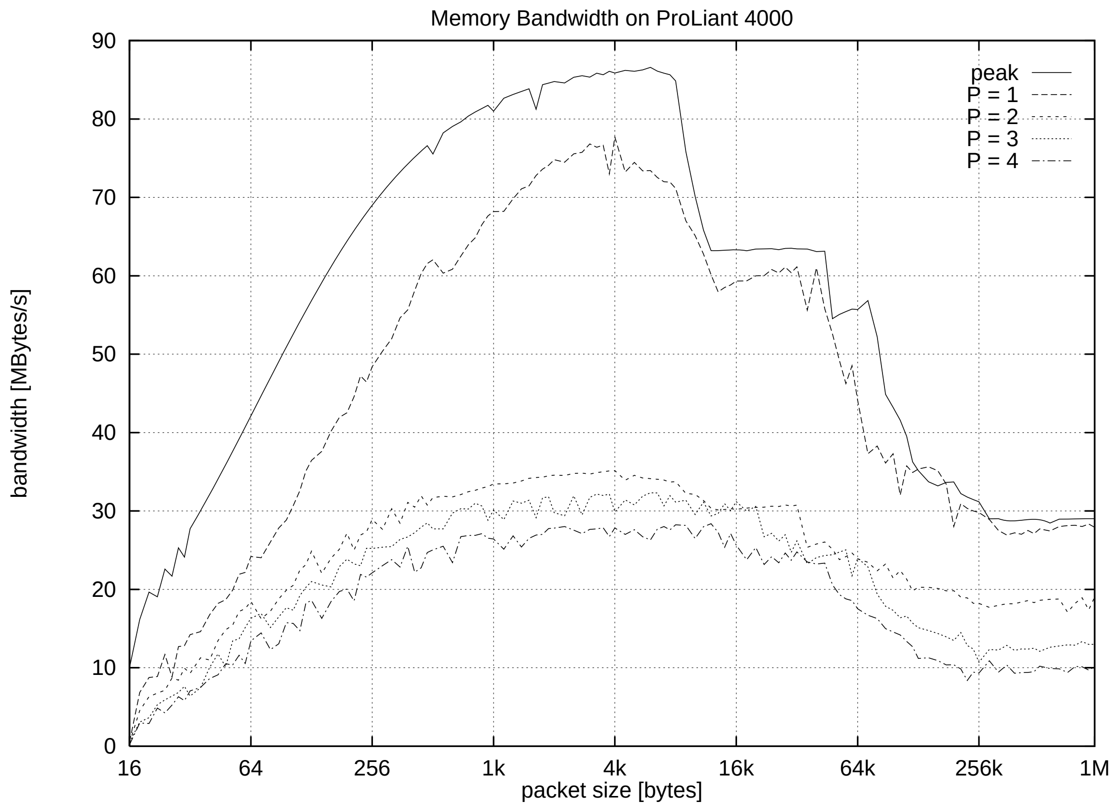


Figure 4: Memory bandwidth on the KSR1 and the ProLiant 4000

The test program does the following:
A packet of fixed size is copied from its source location to the destination. The source array was initialized before by processor *i*, so that its contents is held in the local processor's cache (if the packet fits in). The target array is initialized too, but by processor *i + 1*, so the cache contents of this processor has to be invalidated during the memcpy operation.

Curves for a different number of parallel memcpy'ing processors are shown. On the KSR1 the curves for 2...7 processors are nearly equal (only 2 curves are shown here). That means, the ALLCACHE engine bandwidth is sufficient for all connected processor nodes. This is not the case on the ProLiant 4000: here the processors have to wait until they become the bus master to access the main memory. The more processors are used, the smaller is the achieved bandwidth.

The peak bandwidth curve was obtained over 100 iterations of the memcpy operation, i.e., the data got loaded into the cache at the first access and was fetched from there in all other operations.

The communication delays for small packet sizes are determined by the overhead of the function calls. To compare them with other machines, they are listed as the bandwidth in tables 1 and 2 again (for the single processor case):

packet size [bytes]	bandwidth [MBytes/s]
16	10.00
24	22.59
32	27.73
64	42.15
96	50.93
128	56.86
256	69.00

Table 1: bandwidth for small packet sizes on the KSR1

packet size [bytes]	bandwidth [MBytes/s]
16	0.49
24	0.45
32	0.98
64	2.93
96	3.03
128	3.13
256	4.83

Table 2: bandwidth for small packet sizes on the ProLiant 4000

6 Influence of Memory Latency on Computation

As seen in the previous chapter, the memory bandwidth of the ProLiant 4000 significantly decreases with the number of processors concurrently requesting the memory bus. This fact has some influence on the computation too: if the operands of a numerical operation have to be fetched from main memory, the computation can be delayed due to bus access conflicts.

A test program was written to examine the influence of the memory latency on computation. Several numerical operations were executed on each processor – fully independently from others – with the double vectors a and b and a constant C as operands. The measured execution times were set in proportion to the time on one processor.

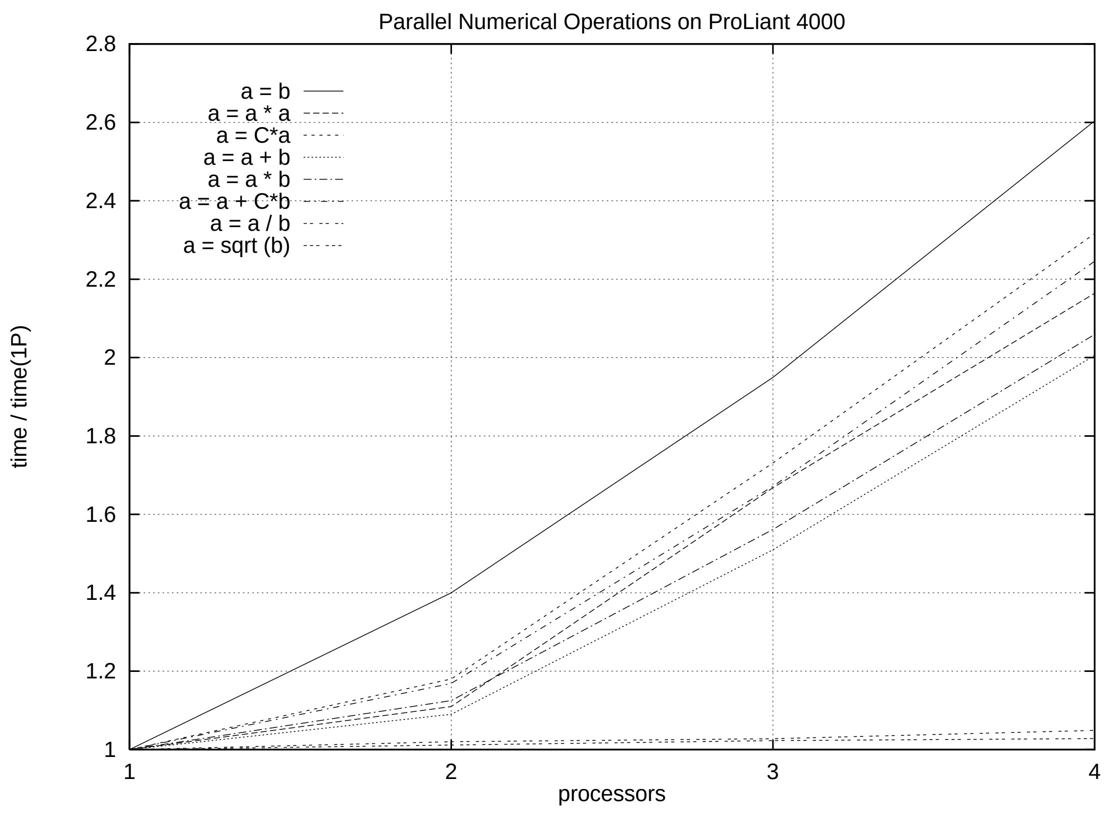


Figure 5: Memory latency influence on independent computations on the ProLiant 4000

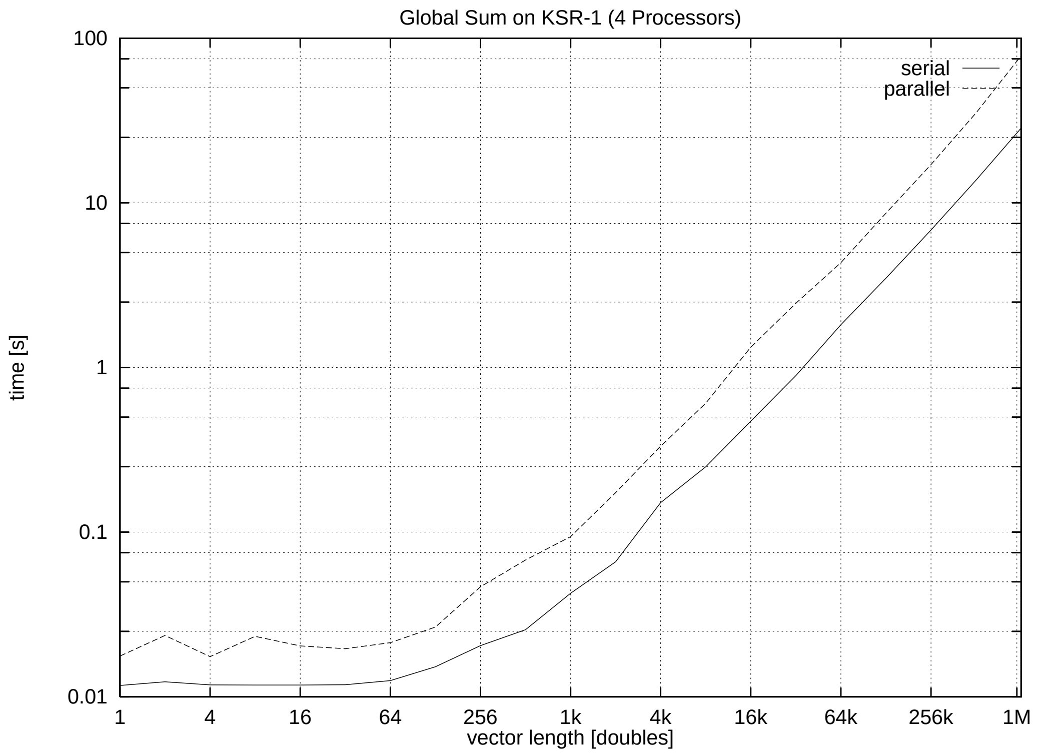
A simple assignment of vector b to vector a takes 2.6 times on 4 processors in contrast to one processor. This operation is analogous to the memcpy curve depicted in the previous chapter. Other numerical operation, like the sum or the product of two vectors, scaling, or elimination, require twice the time of a serial computation. Only in heavyweight calculations (division, square root) the theoretical speedup is reached.

7 Global Operations

To examine the efficiency of global operations, two algorithms for the computation of a global sum of double vectors were implemented:

- a serial version with barrier synchronization
  The first thread that enters the barrier initializes a global operation structure by setting the pointer to its local vector to an intermediate result pointer, and suspends its execution to wait for the other threads. These add their own vector to the intermediate sum vector (the vector of the first thread is used for this) as they reach the barrier, and suspend themselfes too. The last thread computes the final sum vector and then awakenes all suspended threads by a broadcast on the barrier. Now all threads copy the result into its own local result vector.
- a parallel version via hypercube communication
  A hypercube topology is used to exchange the vectors between adjacent nodes. Both the local and the remote vector are added and sent to the next neighbour. This version has a parallelity degree of D (the dimension of the hypercube).

The following diagrams compare both versions on the KSR1 and the ProLiant 4000:



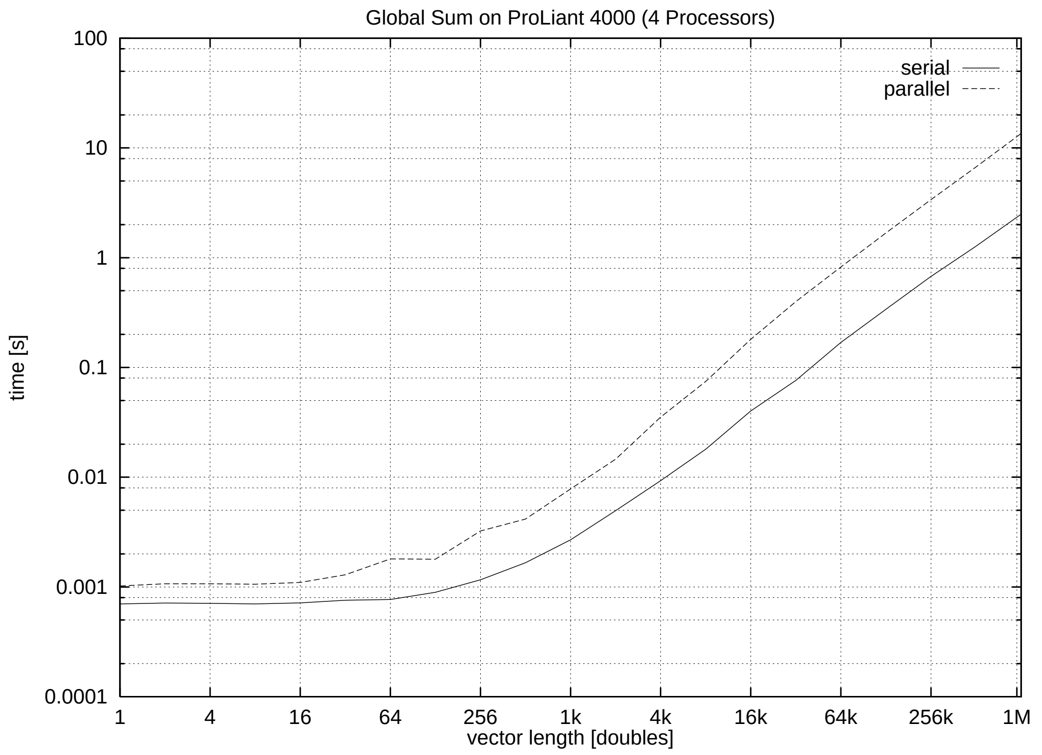


Figure 6: Serial and parallel computation of the global sum on the KSR1 and the ProLiant 4000

As can be seen, the serial algorithm works faster than the parallel one on both machines. There might be two reasons for that: at first, the global sum is a very cheap numerical operation (see figure 6). More complex operation should bring both curves nearer to each other. And secondly, only 4 processors were used (there were not more processors available on the KSR1 to build a hypercube of dimension 3). The more processors calculate in parallel the higher should be the speedup of the parallel algorithm.

8 Conclusions

The communication bandwidth achieved in a message passing emulation on shared memory architectures is mainly determined by their memory bandwidth and therefore is often much higher compared with distributed memory architectures. An important demand to the memory system is scalability: a processor's access to global memory should not be delayed by concurrent independent accesses of other processors. The use of local processor caches significantly increases the speedup, so caches should be as large as possible.

Computations should contain complex operations rather than simple, if the memory bandwidth of the underlying architecture is not sufficient for parallel work of all processors. Otherwise a decrease of speedup is the consequence, even if the operations are fully independently of each other.

For global operations there exists a break-even point, where a serial implementation is still faster than a parallel one (based on the model of true message passing implementations, e.g. hypercube communication). There are very efficient global synchronization methods for threads (e.g. barriers) which can overweigh the parallel versions with their several local synchronizations.

References

[1] T. Radke. *Parallel Programming with User-level Threads*, Technical Report TUCZ / RA-TR-96-3, Computer Science Department, University of Technology Chemnitz, 1996.

[2] L. Grabowsky. *Parallel FEM implementations on shared memory systems*, Technical Report TUCZ / RA-TR-96-4, Computer Science Department, University of Technology Chemnitz, 1996.

[3] *KSR1 Principles of Operations*, Kendall Square Research Corporation, Massachusetts, 1993.

[4] *Pentium^{TM} Processor User's Manual*, Intel Corporation, 1994.