Automatic State Capture of Self-Migrating Computations in MESSENGERS

Christian Wicke[1,2], Lubomir F. Bic[1], Michael B. Dillencourt[1], and Munehiro Fukuda[3]

[1] Information and Computer Science, University of California, Irvine, CA 92697, USA
[2] Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, 76128 Karlsruhe, Germany
[3] Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan

Abstract. With self-migrating computations, the main challenge is the extraction and subsequent restoration of the computation's state during migration. This is very difficult when the navigational statement may be placed anywhere in the code and hence many systems place the burden of state capture on the application programmer. We describe an intermediate approach, where the use of navigational statements is restricted to the top level of the self-migrating computation. This permits an efficient implementation of a fully transparent state capture and restoration. We demonstrate that this approach is applicable not only to interpreted mobile code but also to compiled self-migrating computations executing entirely in native mode.

1 Introduction

Self-migrating computations, commonly also referred to as mobile agents, are self-contained entities that can navigate autonomously through a network and perform various tasks in the nodes they visit. The underlying computational model is essentially a multi-threaded environment, augmented by navigational capabilities. Individual agents consist of a program and a state and are able to communicate with one another via shared or distributed memory mechanisms. Navigational capabilities permit individual computations (i.e., agents) either to move themselves or to send copies of themselves to other nodes in the network so that execution continues in the new environment.

To achieve these navigational capabilities, it is not sufficient to simply spawn a copy of the current computation, since this would cause the spawned copy to start executing from the beginning. Instead, there must be a mechanism for extracting the state of the computation at the source node, restoring it at the destination node, and causing the new or moved instance of the computation to start execution immediately following the computational statement that invoked the navigational capability. Note that this is much more difficult than a context switch. The latter also requires state capture and restoration, however, the state is only that of the processor. In particular, when the mobile agent is compiled and executing in native mode, the state that needs to be saved and restored at context switch is the contents of the hardware registers and flags of the CPU. When the agent is interpreted, its entire context switching state is maintained by the interpreter in memory and thus does not even have to be saved.

For the purposes of migration between different machines the complete state of the computation (not just the processor) must be considered. This includes, in addition to the processor registers and flags, the complete activation stack, any dynamically allocated heap memory, and all file and communication descriptors. Generating the code to automatically extract all of the above information at arbitrary points during the execution and in a manner transparent to the user is extremely difficult, requiring a major extension to the compiler or the interpreter, depending on the agents' mode of execution. Consequently, mechanisms for achieving this have been implemented in only a few systems, such as Telescript [Tel96, Whi94], Agent Tcl [Gra96], and Ara [Pei97, PS97]. Other systems provide a much lower level of support. For example, Java-to-go [LM96] requires the programmer to explicitly save and restore the state during a migration operation. Aglets [LO96] preserve the heap during migration but require that the stack and the program counter be explicitly saved by the programmer.

An intermediate approach to state capture is to restrict the points in the execution at which this can occur. This approach has been used successfully by the MESSENGERS system [BFD96], where the use of any navigational statements is restricted to only the top level of execution, i.e., the equivalent of the main program. It also prohibits the use of pointers at this level. This eliminates the need to extract/restore the activation stack as well as any data on the heap storage and hence only the computation's local variables and its program counter need to be sent along with the code during migration. This not only greatly simplifies the state capture problem but also makes its implementation very efficient.

While earlier versions of the MESSENGERS system were interpreted at the top level, thus making state capture under the above restrictions very simple to implement, we have recently developed a fully-compiled version of MESSENGERS. In this paper, we show that the strategy of limiting the possible locations at which state extraction/restoration can occur leads to a very simple implementation even when the agents are compiled and execute in native mode. The restriction is a very natural one that furthers one of the design goals of MESSENGERS, namely the clean separation of computation from coordination.

After presenting a brief overview of MESSENGERS (Section 2), we describe our approach to state capture in compiled MESSENGERS code in Section 3. In Section 4, we present performance results showing that MESSENGERS is competitive in performance with message-passing approaches to distributed computing.

2 Overview of MESSENGERS

MESSENGERS [BFD96, FBDM98] is a system that supports the development and use of distributed applications structured as collections of autonomous objects, called Messengers[1]. The MESSENGERS system involves three levels of networks. The lowest level is the *physical network* (a LAN or WAN), which constitutes the underlying computational nodes. To allow Messengers to navigate autonomously through the network and carry out their tasks, the MESSENGERS system is implemented as a collection of daemons instantiated on all physical nodes participating in the distributed computation. A daemon's task is to continuously receive Messengers arriving from other daemons, interpret their behaviors, described as programs carried as part of each Messenger, and send them on to their next destinations as dictated by their behaviors. These daemons constitute the middle layer. The *logical network* is an application-specific computation network created on top of the daemon network. At system startup, a single logical node, named *init*, is created on every daemon node. Any Messenger may be injected (from the shell or by another Messenger) into any of the *init* nodes and from these it may start creating new logical nodes and links on the current or any other daemon.

Messenger programs, referred to as *Messenger scripts*, are sequences of statements of the following types:

- *Computational statements* enable the Messenger to perform general computations. They include all standard C assignment and control statements, involving arbitrary variables and constants except pointers.
- *Navigational statements* endow the Messenger with mobility, permitting it to manipulate and move within the logical network. The *hop* statement allows a Messenger to move around the network. Its syntax is as follows:[2]

  $hop(ln = n; ll = l; ldir = d)$

  where ln stands for "logical node", ll stands for "logical link", and $ldir$ stands for the link's direction. Other commands allow a Messenger to create and delete nodes and links in the logical network.
- *Function invocation statements* provide an interface to the system's environment. They permit the dynamic loading and invocation of arbitrary precompiled functions written in unrestricted C and executed in native mode.
- *Synchronization statements* allow Messengers to wait for certain conditions to occur and to signal the occurrence of conditions.

Given the MESSENGERS system's orientation toward general-purpose high-performance computing, the following design choices have been made in the compiled version:

1. An executing Messenger can only be preempted when it performs a navigational statement, a (blocking) synchronization statement, or voluntarily

[1] The individual autonomous objects are denoted by mixed case (Messengers), while the system as a whole is denoted by small capitals (MESSENGERS).

[2] The syntax shown here is somewhat simplified. In its full generality, a single *hop* statement supports multiple hop specifications and an elaborate set of options as specified by the navigational calculus described in [FBDM98].

gives up the processor. Hence each sequence of instructions between any such statements is automatically an uninterruptible critical section.

2. There are two types of variables: node variables, which are stationary in every logical node, and messenger variables, which are carried by a Messenger. This clearly separates a node state from a messenger state.
3. The distinction between the code at the script level of every Messenger and the C functions it invokes, provides for a clear separation between the coordination and the computation layers of an application.

These have important implication for the problem of state capture, as described below.

3 State Capture with Compiled Messengers

As described in Section 2, the current version of MESSENGERS uses a combination of interpreted code at the top (script) level while all invoked functions execute in native mode. The interpretation overhead is significant and hence a fully compiled version is currently being developed. This means that each Messenger program is compiled entirely into native code. The main difficulty is the implementation of the statements that may cause a context switch. As mentioned above, this includes (1) navigational statements, (2) blocking synchronization statements, and (3) statement that voluntarily give up the processor (including the termination of a Messenger). For each of these cases, the state of the computation must be captured and saved automatically and transparently in a data structure. In the case of a navigational statement, this data structure is sent to another node where the Messenger is recreated and continues executing. In the other two cases, the Messenger's execution is interrupted and continues on the same logical node at a later time.

As already explained in Section 1, restricting the placement of all context-switching statements to the top level of the Messenger script guarantees that the state of the computation consists of only the local variables and the program counter. This information is kept in a data structure, called the *Messenger control block*, which is maintained for each Messenger at its current logical node. To perform a navigational statement, the local variables can easily be extracted and packed into a message for transport. The main difficulty is the program counter, which cannot be carried in the form of a memory address between machines.

Our solution, which handles all context-switching statement in a uniform way, is to restructure the Messenger script into a collection of *script-level functions* as follows:

1. All statements of the original script become part of one of the new functions; that is, the original script consists of only function calls.
2. All functions are numbered and each knows (at compile time) its possible successors, i.e., the numbers of the functions to execute next. Note that, due to conditional and loop control statements, a function may have more than one possible successor, which is selected at runtime.

```
S1;
SC;
S2;
...
```

(a)

```
f1(mcb) {                f2(mcb) {
  S1;                      S2;
  mcb->next_func=2;        mcb->next_func=...;
  //code for SC//;       }
}
```

(b)

Fig. 1. Transformation of straight-line code

3. Any statement that may cause a context switch may execute only as the *last* statement of a function. This guarantees that we only need to capture the state *between* functions. Hence instead of the actual program counter, the state can simply record the number of the next function to be executed. This is a machine-independent integer, called *next_func*, and is maintained in each Messenger control block.

Figure 1 illustrates the above transformations for straight-line code. Part (a) of this figure shows a fragment of MESSENGERS code containing a context-switching statement SC, such as a *hop*, preceded by a sequence ($S1$) of other statements and followed by another sequence of statements ($S2$).

Part (b) shows the pseudo code for the two functions into which this script is transformed. Both functions are passed a pointer, *mcb*, to the Messenger control block as a parameter. Function $f1$ starts with the sequence of statements $S1$. It then assigns 2 as the number of the function to be executed next. (This is used as an index and identifies the function $f2$.) Finally, it executes the code generated by the compiler for the statement SC. In the case of a navigational statement (e.g., *hop*), this code would extract the state of the Messenger from its control block and send a copy to all destinations as specified by the *hop* parameters.

As a result of this send, the function $f2$ is started on all nodes receiving the state. This executes the statements $S2$ and, prior to exiting, sets the value of the *next_func* variable.

If SC is not a navigational statement but a (local) synchronization statement, the code generated for this statement would dequeue the Messenger control block from the ready list and enqueued it at the appropriate waiting list. When it is later returned to the ready list, it continues with $f2$ as the next function.

The above transformations may be applied when the context-switching statements are at the top level of the script, that is, outside of any conditional and loop constructs. To satisfy the requirement that any context switching statement may execute only as the last statement of any function (condition 3 above), additional transformations must be applied, which split conditional and loop constructs into multiple functions along any embedded context-switching statements.

Figure 2 illustrates the transformation of a conditional (if-then-else) statement, where both branches contain a SC statement. To show how the conditional is integrated with its enclosing statements, the conditional shown in part (a) is preceded by a sequence of statements $S1$ and succeeded by a sequence S6.

```
S1;
if (...) {
  S2;
  SC;
  S3;
} else {
  S4;
  SC;
  S5;
}
S6;
```

(a)

```
f1(mcb) {
  S1;
  if (...) {
    S2;
    mcb->next_func=2;
    //code for SC//;
    return;
  } else {
    S4;
    mcb->next_func=2;
    //code for SC//;
    return;
  }
}

f2(mcb) {
  S3;
  S6;
}

f3(mcb) {
  S5;
  S6;
}
```

(b)

Fig. 2. Transformation of conditional code

```
while (...) {
  S1;
  if (...) {
    S2;
    SC;
    S3;
  } else S4;
  S5;
}
```

(a)

```
f1(mcb) {
  while (...) {
    S1;
    if (...) {
      S2;
      mcb->next_func=2;
      //code for SC//;
      return;
    } else S4;
    S5;
  }
}

f2(mcb) {
  S3;
  S5;
  while (...) {
    S1;
    if (...) {
      S2;
      mcb->next_func=2;
      //code for SC//;
      return;
    } else S4;
    S5;
  }
}
```

(b)

Fig. 3. Transformation of a while loop

Part (b) shows the three functions resulting from this script. Each is again passed the pointer to the Messenger control block, *mcb*, as a parameter. Function $f1$ starts with the statements $S1$ and then includes the code of both branches that precede the SC statements. The then-branch sets *next_func* to 2, followed by the code generated for the SC statement. Similarly, the else-branch sets *next_func* to 3, followed by analogous code for SC. Consequently, when the Messenger resumes its execution after the SC statement, one of the functions $f2$ or $f3$ is invoked as the successor, depending on the current *next_func* value. This completes the corresponding branch and continues with the sequence $S6$.

Figure 3 illustrates the transformation of a while-loop containing a SC statement. This statement is embedded inside a conditional statement (part (a)). Part

(b) shows the two resulting functions. Function $f1$ contains the entire while-loop except for the code following the SC statement (i.e., the sequence $S3$). Instead, this branch ends with setting *next_func* to 2, followed by the code for SC. The successor function, $f2$, then contains the remainder of the path, $S3$, which is executed when the Messenger resumes execution after the SC. The sequence following $S3$ is $S5$, which ends the original while-loop. The while-loop is then replicated to continue the next iteration. Note that, instead of replicating the while-loop, function $f2$ could simply return $f1$ as its successor and terminate. The replication guarantees that no unwanted context switch will occur in the middle of the while loop. It also gives the compiler a larger window of code for better optimization. We expect this to outweigh the cost of replication, since the Messenger script level is the "coordination" level of the computation, where deep nesting of loops or conditionals is not likely to occur.

The above compilation approach guarantees that every Messenger program is a simple *linear sequence of function invocations, without any conditionals, iterative statement, or other control statements.* This permits us to replaces the current interpreter daemon in each physical node by a new daemon, which maintains the ready lists of all Messengers and repeatedly performs the following tasks:

- For all Messengers currently on the ready list, invoke the function identified by its *next_func* value; before it terminates, this function automatically updates the *next_func* value with the number of its successor function, as described above. Note that state capture as well as any movement of Messengers between the ready queue and other queues is not performed by the daemon but by the code generated by the compiler for the corresponding context-switching statement.
- After a complete pass through the ready list, receive any messages (each containing a Messenger control block) arriving from other nodes and enqueue these at the ready list.

The above discussion ignores the issue of I/O connections. Since MESSENGERS can run on any network, maintaining the status of file connections may not be meaningful if a Messenger hops between physical nodes that do not share a common file system. However, in the special case where a hop is between two physical nodes that access a common file, it is useful to preserve a file connection across navigational statements. This can be achieved by providing special MESSENGERS functions to open and close files instead of the standard C library functions. This permits the MESSENGERS system to keep track of all files opened by each Messenger. All the information needed to restore the file status (filename, position pointer, access permission) may be added to the messenger state. After moving to a new physical node, the files can then be reopened transparently in the same state as before the navigational statement.

4 Matrix Multiplication

In this section we contrast a distributed application using compiled MESSENGERS with an implementation of the same application using a message-passing approach. The application we have chosen is matrix multiplication. We selected this applications for two main reasons. First, matrix multiplication is an essential component of a very large class of numerical computations and hence is an important problem in its own right. Second, MESSENGERS is intended for general-purpose computing. Many scientific problems either use matrix multiplication directly or must solve similar numeric problems that, like matrix multiplication, are highly regular in their structure and require significant movement of data between computations. Hence matrix multiply represents a good test case for studying MESSENGERS' capabilities in this arena. This includes both its ability to describe the problem intuitively in a distributed manner and the resulting performance. We compare the MESSENGERS implementation with am implementation using PVM [GBD+94]. PVM was chosen because it is a widely available, commonly used system that supports the message-passing approach to distributed computing. We compare performance of the original interpreted MESSENGERS, th new compiled MESSENGERS, PVM, and sequential implementations.

The simplest possible implementation of matrix multiplication on a single processor is a triply nested loop, where the outer and middle loops iterate over the rows and columns of the two matrices and the inner-most loop (k) computes the inner product of a row and a column as follows: $C[i,j] = C[i,j] + A[i,k] * B[k,j]$. This naive implementation of matrix multiplication is adequate when the matrices are small. Considerable effort has gone into developing sequential algorithms that are faster for large matrices but more complex to program; see for example [Pan84, CW90]. Much of the emphasis in this research has been improving the asymptotic performance beyond the $\Theta(n^3)$ running time of the naive algorithm for $n \times n$ matrices. Even modifications that do not improve the asymptotic performance beyond $\Theta(n^3)$ can provide some useful speedup. For example, partitioning a matrix into smaller blocks and then decomposing the multiplication into a series of additions and multiplications of these blocks can result in some speedup, because multiplying the smaller sub-matrices obtained by partitioning increases cache utilization and reduces paging overhead. The precise amount of speedup depends on the specific machine architecture. Our experiments indicate that on a 110 MHz SPARCstation 5 with 32MB of memory, partitioning a 1500×1500 matrix into 9 blocks of size 500×500 results in a speedup of roughly 13%.

The main point is that the sequential algorithm can be improved by carefully orchestrating data accesses to the matrices. In general, more speedup requires more careful orchestration, which significantly increases the complexity of implementing the algorithm. When compared with parallel matrix multiplication algorithms, we observe that these are no more difficult to write than those optimized for a single processor, yet we gain the additional benefit of speedup by utilizing multiple processors. In the remainder of this section we consider a wellknown algorithm for parallel matrix multiplication [GBD+94], which we implement using two very different approaches to parallel program construction—one using message-passing and the other using MESSENGERS.

The algorithm is block-oriented. The two $m \times m$ input matrices A and B, and the resulting matrix C are all partitioned into rectangular blocks of size $s \times s$. We will use $A[i,j]$ to denote the corresponding $s \times s$ block of matrix A, i.e., all elements of A with indices ranging from $(i * s)$ to $(i * s + s - 1)$ and $(j * s)$ to $(j * s + s - 1)$. Similarly, $B[i,j]$ and $C[i,j]$ denote the corresponding $(s \times s)$ blocks of B and C. Each block $A[i,j]$, $B[i,j]$, and $C[i,j]$ is assigned to a different processor, addressed using the same coordinates $[i,j]$. The blocks of C remain stationary, that is $C[i,j]$ always resides on processor $[i,j]$, while the blocks of A and B are moved between the processors according to the following three distinct phases:

1. *Distribution of A*: During each iteration, k, all blocks $A[i,j]$ whose indices satisfy the condition $j = (i + k) \bmod s$ are multi-cast to all processors in the same row. During iteration 0 all elements on the diagonal $A[0,0]$, $A[1,1]$, ..., $A[m-1,m-1]$ are distributed to each row. The diagonal to be distributed is then shifted one to the right. That is, during iteration 1 the elements $A[0,1]$, $A[1,2]$, ..., $A[m-1,0]$ are distributed, etc.

2. *Block multiplication*: The block of A received during the previous step is multiplied with the block B currently residing in the processor and is added to the block of C assigned to the processor.

2. *Rotation of B*: Each block of B is moved to the neighboring processor in the same column using a circular shift. That is, blocks of B residing on processors $[i,j]$ are shifted to processors $[(i-1) \bmod s, j]$.

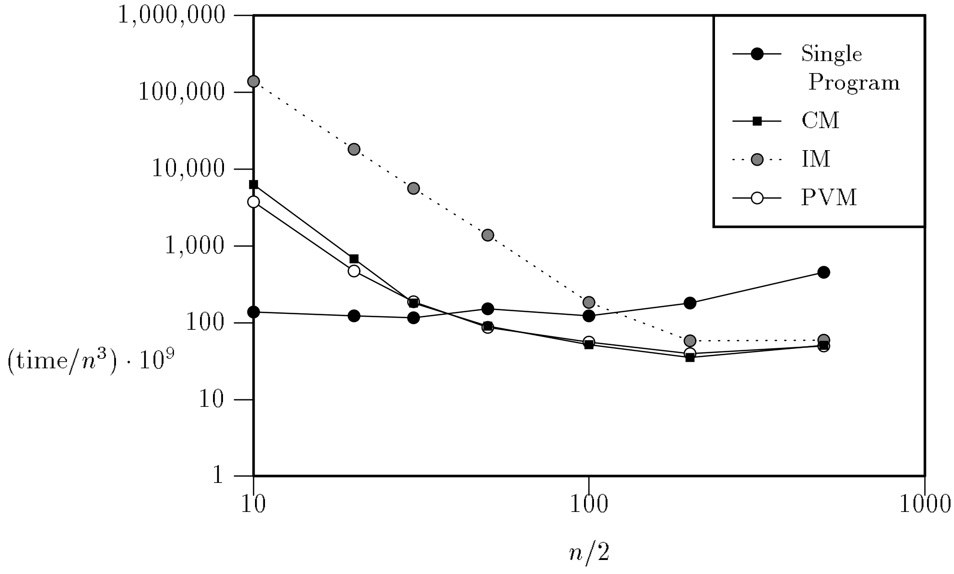
After repeating the above three phases m times, the (distributed) matrix C contains the result of the complete multiplication of the matrices A and B.

4.1 Performance

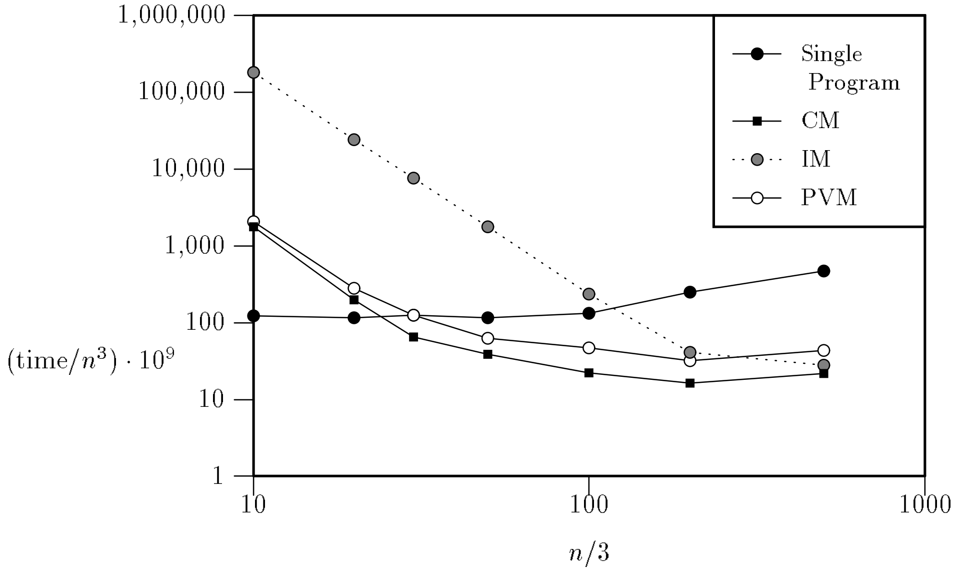
We have conducted the performance comparison between interpreted MESSENGERS, compiled MESSENGERS, PVM, and sequential C implementations on a 2×2 and a 3×3 grid of processors, both using an Ethernet-based LAN of SPARCstations 5. All experiments were run on 170 MHz SPARCstations. The results are shown in Figure 4(a) and (b), The experiments on the 2×2 grid were run for matrices of size 20×20, 40×40, 60×60, 100×100, 200×200, 400×400, and 1000×1000. Thus the block sizes were, respectively, 10×10, 20×20, 30×30, 50×50, 100×100, 200×200, and 500×500.

The experiments on the 3×3 grid were run with the same set of block sizes, so the matrices were of size 30×30, 60×60, 90×90, 150×150, 300×300, 600×600, and 1500×1500. The various programs benchmarked are as follows:

- *Single Program:* A straightforward sequential matrix multiplication, with no partitioning of the matrix.
- *CM:* Compiled Messenger, as discussed in this paper.
- *IM:* The earlier (interpreted) version of Messenger.



(a)



(b)

Fig. 4. Matrix Multiplication results. (a) On 4 processors arranged as a 2×2 grid. (b) On 9 processors arranged as a 3×3 grid.

– *PVM:* The PVM matrix multiplication program discussed in this section. In the PVM implementation, the row is a group and requires a group server, which is executing on its own machine.

As can be seen from the figures, the compiled MESSENGERS version is comparable with PVM running with an additional machine for the group server (indeed, it is faster for a 3×3 grid). The compiled version of Messenger is considerably faster than the interpreted version for smaller block sizes, but not for larger block sizes. The reason for this is that the interpreted version calls a native mode C function to do the copying and multiplication of blocks, so that as the blocks get larger, the fraction of the time that the interpreted version spends interpreting Messenger code (as opposed to running native C code) decreases.

5 Conclusion

State capture is the process of extracting the state of a computation so that it may be moved transparently to a new physical node and continue executing in the new environment. Fully automatic state capture can be difficult to accomplish when the navigational statement may be placed anywhere in the code. We have described an intermediate approach, where the navigational statements are restricted to the top level of the self-migrating computation. This not only preserves the separation between the computational layer of the application, which consists of an arbitrary collection of functions distributed throughout the system, and the coordination layer of the application, which deals primarily with the locations at which various functions are to be executed and the communication among them; it also permits a very efficient implementation of fully transparent state capture and restoration during migration. This approach has been implemented in the latest version of our MESSENGERS system.

We contrasted the mobile agent approach to general-purpose distributed programming, as typified by MESSENGERS, with the conventional message-passing approach in the context of a specific application, matrix multiplication. The compiled MESSENGERS system achieves significant speedup over sequential C programs once the problem size is sufficiently large. Not surprisingly, it also shows a dramatic improvement over the earlier interpreted version of MESSENGERS. Finally, it gives a slightly better performance than an implementation using conventional message-passing approaches as typified by PVM.

For additional information the interested reader is invited to browse our WWW page: `http://www.ics.uci.edu/~bic/messengers`.

References

[BFD96] L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8), Aug. 1996.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[FBDM98] M. Fukuda, L.F. Bic, M. Dillencourt, and F. Merchant. Distributed coordination with messengers. *Science of Computer Programming*, 31(2), 1998. Special Issue on Coordination Models, Languages, Applications.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jieng, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[Gra96] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996. `http://www.cs.dartmouth.edu/~agent/papers/index.html`.

[LM96] W. Li and D.G. Messerschmitt. Java-to-go. Technical report, Dept. of EECS, University of California, Berkeley, 1996. `http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go/`.

[LO96] D. B. Lange and M. Oshima. Programming mobile agents in Java with the Java Aglet API, 1996. `http://www.trl.ibm.co.jp/aglets/`.

[Pan84] V. Pan. *How to multiply matrices faster.* Lecture Notes in Computer Science 179. Springer-Verlag, Berlin, 1984.

[Pei97] H. Peine. An introduction to mobile agent programming and the Ara system. ZRI Technical Report 1/97, Dept. of Computer Science, University of Kaiserslautern, January 1997. `http://www.uni-kl.de/AG-Nehmer/Ara/ara.html`.

[PS97] H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proc. of the First International Workshop on Mobile Agents (MA'97)*, Berlin, Germany, April 1997. Springer Verlag, Lecture Notes in Computer Science No. 1219. `http://www.uni-kl.de/AG-Nehmer/Ara/Doc/architecture.ps.gz`.

[Tel96] The Telescript reference manual. Technical report, General Magic, Inc., Mountain View, CA 94040, June 1996. `http://www.genmagic.com`.

[Whi94] J.E. White. Telescript technology. Technical report, General Magic, Inc., Mountain View, CA 94040, 1994.