Optimizations for Automatic Extraction of Parallelism

Antonio Magnaghi and Hidehiko Tanaka
The University of Tokyo

1 Introduction

In scientific research world, massively parallel comput-
ers are considered to be one of the most promising an-
swers to the remarkable demand increase in computa-
tional power. The continuos improvements in hardware
technologies had made possible to produce computers
based on articulated and complex architecture, indeed
capable of a high system performance on a potential
level. However, a frontier challenge is to effectively
exploit available resources in order to bridge the gap
between actual and achievable performance. The pro-
cess of efficient parallel software development becomes
a critical factor, being extremely error-prone because
of the inherent complexity of the programming envi-
ronment. Parallelizing compilers [2, 3] for imperative
languages aim at automatically discovering as much
parallelism as possible from sequential source code, and
at mapping the original program to a modified paral-
lel one with improved execution performance. Recon-
struction techniques have been extensively developed
for vector computers, but in the field of massively par-
allel machines relevant questions still need analysis.

2 Compiler implementation design

The implementation part of our project is concerned
with the design of a source-to-source compiler (fig.1).
It will perform the fundamental optimizing mappings
on input code, producing as output a program to be
executed on a massively parallel computer. In addi-
tion to this, our goal is to study reconstruction criteria
that might allow the extraction of more concurrency
on a parallel computer. Our research activity aims at
designing a parallelizing compiler that uses imperative
C-like source code as input to produce optimized par-
allel code as output. The compiler implementation de-
sign considers that the abstraction level of the analyzed
source code has a strong impact on the kind of opti-
mizations to perform. Therefore, a clear distinction is
introduced between high level and low level mappings.
This approach well matches the typical modular orga-
nization of commercial compilers into a front-end and
a back-end.

The first group of reconstruction techniques, to be
included in the front-end of the compiler, relays on
machine independent properties of the input code in
order to increase its portability on different systems.
As a first step, standard optimizing transformations
presented in the literature are being implemented. The
compiler front-end gathers information about control
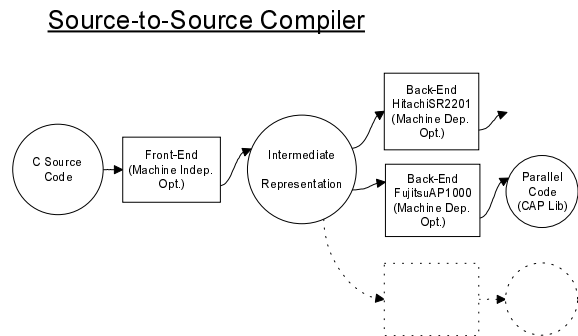flow and dependency analysis to be conveniently

Source-to-Source Compiler

Figure 1: Compiler functional structure

exploited in successive transformations. An interme-
diate code representation is produced and optimiza-
tions are carried out including constant propagation,
loop normalization [3] and induction variables identifi-
cation [2]. In some cases a variety of transformations
might be applicable, but some of them can later inhibit
successive reconstruction phases. Thus, additional in-
formation needs to be produced, because it is necessary
to address the issue of "mappings invertibility", in or-
der to undo code transformations, if convenient.

On the other hand, the machine dependent group of
restructuring tools, to be included as part of the back-
end, addresses more specific transformations. Concur-
rency is to be extracted mainly from loops contained in
the program, basing on data dependency and control
flow analysis. In this context, program transformations
are more strictly linked to architectural characteristics
of the target computer. For instance, improving mem-
ory locality through loops reordering and strip-mining
requires additional information about the cache lines
size, data partitioning in shared memory space will re-
sult in execution speed-up only if synchronization and
communication overhead costs are conveniently evalu-
ated.

The adopted strategy is to design a parametric back-
end for distributed memory computers, basing on pe-
culiar architectural parameters. Therefore it is pos-
sible to build a compiler for different computers by
re-using the portable front-end and by modifying the
set of parameters the back-end employs. In partic-
ular, our interest is in the massively parallel com-
puter *HitachiSR2201* and in *FujitsuAP1000*. Both
have distributed-memory and are based on a message-
passing paradigm. Target code performs message pas-
sage through MPI (Message Passage Interface) library
on HitachiSP2201, and through CAP library on Fujit-
suAP1000.

As concerns the compiler life cycle, an incremental approach seems to be the most valuable, in order to develop successive extended versions with enlarged sets of optimizing instruments. Hence, new optimizing algorithms can be effectively integrated with the core of the compiler itself, enabling a comparative evaluation of performance improvement and a more complete understanding of key factors in parallel computation.

3 Preliminary considerations for parallelism extraction

Parallel execution requires data distribution and duplication on computer nodes, because of dependencies among different program statements. As a consequence, consistency issues arise. It is critical to decide how to map data, because on one hand generally many different alternatives are available, and on the other hand the adopted choice will have a remarkable impact on the global amount of communication. Therefore some implemented systems require a direct interaction with the programmer.

A possible direction we are interested in investigating is to perform a data partition based on dependency constraints in order to identify distinct code points from which parallel execution could independently start. Considerations about usage of data in the program computations could pinpoint those portions more weakly or not even interfering with each other in sharing information. Thus, improvements can derive from limiting locally duplicated data to strictly necessary parts and minimizing communication costs among computer nodes. In this context, it is worthwhile how to properly represent data dependency. FUD (Factored Use Definition [2]) chains are a powerful and compact manner to link a variable use to its reaching definitions, taking into account at the same time also control flow structure of the analyzed program. The analysis of FUD chains for some code fragments seems to help understand better the characteristics of the algorithm to compile. Basing on the following simple example, we are aiming at intuitively showing the reason why a proper data partition on FUD chain can produce a positive impact on execution time.

```
for(I=0;I<N;I++)
        {
        S1 : b=a[I]+1;
        S2 : c = a[I]-b;
        S3 : d = c+200;
        S4 : f = a[I]/2;
        S5 : a[I] = f+2;
        S6 : a[I+1] = f+250+a[I];
        }
```

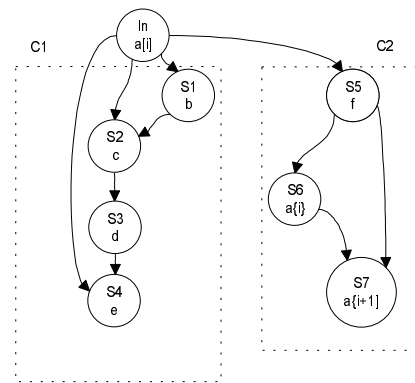Figure 2 presents the FUD chain for the loop body. We can distinguish two graph components

Figure 2: FUD chain representation

C1={In,S1,S2,S3,S4} and C2={In,S5,S6,S7} employing as input a[I]. The element a[I] is the unique common node for C1 and C2. Eliminating it produces a disconnected graph.

Each loop iteration could be executed faster by the employment of two processors P1 and P2, one for component C1 and one for C2, requiring duplication of only a[I] on P1 and P2. Moreover, we can get an additional improvement through parallel execution compared to sequential one, because such a decomposition allows to hasten next iteration. The i-th loop iteration can be started as soon as the value of a[I] is available. Therefore, through the considered partition it is also possible to execute in advance the (i+1)-th iteration, successive to i-th one. In the sequential case, the computation of a[I+1], input element to (i+1)-th iteration, requires the value of a[I] in S6. However S6 can be executed only after S4 because of an anti-dependence constraint from S4 to S6. As a consequence a[I+1] is to be produced at the end of each iteration, therefore the execution of the successive iteration is to be delayed until the previous one is over. In the parallel case, the value of a[I+1] is determined earlier, on processor P2, independently from processor P1, hence the next (i+1)-th loop iteration can begin before the i-th one is terminated.

4 Future work

Design activity is driving the programming phase in order to implement and test the optimizing compiler. Then, a comparative analysis will be carried out on commercial benchmarks.

References

[1] T. Fahringer
Automatic performance prediction of parallel programs. Kluer Academic Publ., 1996

[2] M. Wolfe
High performance compilers for parallel computing. Addison-WesLey, 1996

[3] H. Zima, B. Chapman
Supercompilers for parallel and vector computers. ACM Press, Addison-Wesley, 1991